Department of AERONAUTICS and ASTRONAUTICS
STANFORD UNIVERSITY

Darpa/Navy Contract No. N00014-92-J-1809

ControlShell: A Real-Time Software Framework

Final Report - July 1995

Stanford University:

Professor Robert Cannon
Professor Jean-Claude Latombe

RTI subcontract:

Dr. Stan Schneider

DEPARTMENT OF THE NAVY
OFFICE OF NAVAL RESEARCH
SEATTLE REGIONAL OFFICE
1107 NE 45TH STREET, SUITE 350
SEATTLE WA 98105-4631

IN REPLY REFER TO:

4330
ONR 247
11 Jul 97

From: Director, Office of Naval Research, Seattle Regional Office, 1107 NE 45th St., Suite 350, Seattle, WA 98105
To: Defense Technical Center, Attn: P. Mawby, 8725 John J. Kingman Rd., Suite 0944, Ft. Belvoir, VA 22060-6218

Subj: RETURNED GRANTEE/CONTRACTOR TECHNICAL REPORTS

1. This confirms our conversations of 27 Feb 97 and 11 Jul 97. Enclosed are a number of technical reports which were returned to our agency for lack of clear distribution availability statement. This confirms that all reports are unclassified and are "APPROVED FOR PUBLIC RELEASE" with no restrictions.

2. Please contact me if you require additional information. My e-mail is *silverr@onr.navy.mil* and my phone is (206) 625-3196.

ROBERT J. SILVERMAN

Darpa/Navy Contract No. N00014-92-J-1809

ControlShell: A Real-Time Software Framework

Final Report – July 1995

PRINCIPAL INVESTIGATORS:

- Stanford University:
  - Robert Cannon, cannon@sierra.stanford.edu, (415) 723 3601
  - Jean-Claude Latombe, latombe@cs.stanford.edu, (415) 723 0350
- RTI subcontract:
  - Stan Schneider, stan@rti.com, (408) 720-8312

Executive Summary

We have created a new paradigm for building and maintaining complex real-time software systems for the control of moving mechanical systems. This objective has been met through the *simultaneous* development of both a powerful software environment and cogent motion planning and control capabilities. Our research has concentrated on three key areas:

- Building an innovative, powerful real-time software framework,
- Implementing new distributed control architectures for intelligent mechanical systems, and
- Developing distribution architectures and new algorithms for the computationally "hard" motion planning and direction problem.

This research built on ControlShell, originally developed at the Stanford Aerospace Robotics Laboratory and now marketed by Real-Time Innovations, Inc. The research added fundamental new capabilities, including network-extensible data flow control to allow scalable support for distributed systems and a graphical environment to enable rapid prototyping and increased productivity. The concurrent development of this tool and experimental robotic applications was aimed at ensuring high-quality architectural design and producing reusable components.

The research was also aimed at developing more specific modules, mainly motion planning modules, which can operate under time constraints. Since motion planning is a computationally hard problem, one cannot expect to achieve strict real-time motion planning. However, one may hope that

future motion planners will ultimately provide a good on-line service by distributing their computation appropriately. Distribution of computation is the central theme underlying our research on motion planning.

Perhaps more importantly, we are working on the *vertical integration* of these technologies into a powerful, working system. It is only through this coordinated, cooperative approach that a truly revolutionary, usable architecture can result.

Chapter 1

Overview

The goal of this research project is to build a new paradigm for building and maintaining complex real-time software systems for the control of moving mechanical systems. This objective is being met through the *simultaneous* development of both a powerful software environment and cogent motion planning and control capabilities. Our research has concentrated on three areas:

- Building an innovative, powerful real-time software development environment,
- Implementing a new distributed control architecture, and using it to deftly control and coordinate real mechanical systems, and
- Developing a computation distribution architecture, and using it to build on-line motion planning and direction capabilities.

We believe that no technology can be successful unless proven experimentally. We are thus validating our research by direct application in several disparate, real-world settings.

This concurrent development of system framework, sophisticated motion planning and control software, and real applications insures a high-quality architectural design. It will also embed, in *reusable* components, fundamental new contributions to the science of intelligent motion planning and control systems. Researchers from our three organizations, the Stanford Aerospace Robotics Laboratory (ARL), the Stanford Computer Science Robotics Laboratory (CSRL), and Real-Time Innovations, Inc. (RTI) have teamed to cooperate intimately and directly to achieve this goal. The potential for advanced technology transfer represented by this cooperative, vertically-integrated approach is unprecedented.

Framework Development This research builds on an object-oriented tool set for real-time software system programming known as *ControlShell*. It provides a series of execution and data interchange mechanisms that form a framework for building real-time applications. These mechanisms

are specifically designed to allow a component-based approach to real-time software generation and management. By defining a set of interface specifications for inter-module interaction, ControlShell provides a common platform that is the basis for real-time code exchange and reuse.

Our research is adding fundamental new capabilities, including network-extensible data flow control and a graphical CASE environment.

Distributed Control Architecture This research combines the high-level motion planning component developed by the previous effort with a deft control system for a complex multi-armed robot. The emphasis of this effort is on building interfaces between modules that permit a complex real-time system to run as an interconnected set of distributed modules. To drive this work, we are building a dual-arm cooperative robot system that will be able to respond to high-level user input, create sophisticated motion and task-level plans, and execute them in real time. The system will be able to effect simple assemblies while reacting to changing environmental conditions. It combines a world modelling system, real-time vision, task and path planners, an intuitive graphical user interface, an on-line simulator, and sophisticated control algorithms.

Computation Distribution Architecture This research thrust addresses the issues arising when computationally complex algorithms are embedded in a real-time framework. To illustrate these issues we are considering two particular problem domains: *object manipulation by autonomous multi-arm robots* and *navigation of multiple autonomous mobile robots in an incompletely known environment.* These two problems raise a number of generic issues directly related to the general theme of our research: motion planning is provably a computationally hard problem and its outcomes, motion plans, are executed in a dynamic world where various sorts of contingencies may exist.

The ultimate goals of our investigation are to both provide real-time controllers with on-line motion reactive planning capabilities and to build experimental robotic systems demonstrating such capabilities. Moreover, in accomplishing this goal, we expect to identify general guidelines for embedding a capability requiring provably complex computations into a real-time framework.

1.1 Main Accomplishments

The accomplishments of our research were reported on a quarterly basis in our intermediate reports. Below we give a synthesis of these accomplishements. The following chapters will describe the most important ones in greater detail.

1.1.1 ControlShell Framework Development

The development of the total control system for sophisticated real-time applications presents significant software challenges. Such systems are usually developed from scratch by a team of programmers with precious little reuse of off-the-shelf code. The notorious difficulty of this programming task makes these systems expensive and hard to maintain and upgrade. ControlShell is a next-generation CASE "framework" developed to facilitate and encourage component-based software development and reuse. ControlShell's well-defined structure, graphical tools, and data management provide a unique component-based approach to real-time software generation and management. ControlShell is designed specifically to enable modular design and implementation of real-time software. By defining a set of interface specifications for inter-module interaction, ControlShell provides a basis for real-time code development and exchange.

ControlShell includes many system-building tools, including a graphical data flow editor, a component data requirement editor, and a state-machine editor. It also includes a distributed data flow package, an execution configuration manager, a matrix package, and an object database and dynamic binding facility. ControlShell is being used in several applications, including the control of free-flying robots, underwater autonomous vehicles, and cooperating-arm robotic systems.

ControlShell was developed as a result of this contract and has been released into a commercial product that is dominating the research robotics market.

1.1.2 Distributed Control Architecture

Distributed applications that require many-to-many communications present a significant development challenge. More so when real-time communications requirements and changing communication patterns are imposed as additional constraints. Distributed control systems is one application area where this problems are often present. The Network Data Delivery Service (NDDS) is a novel network data-sharing system developed to address the needs of distributed control applications. NDDS builds on the model of information producers (sources) and consumers (sinks). Producers generate data at their own discretion, unaware of prospective consumers. Consumers "subscribe" to data-updates without concern for who is producing them. The routing protocol is connectionless and nearly "stateless," thus network reconfigurations, node failures, etc. are handled naturally. This scheme is particularly effective in systems (such as distributed control systems) where information is of a repetitive nature.

NDDS provides support for multiple producers, reliable data-delivery, consumer update guarantees, notifications vs. polling for updates, dynamic binding of producers and consumers, distributed queries, user-defined data types etc. NDDS is integrated into the *ControlShell* real-time framework.

NDDS was developed as a result of this contract and has been released into a commercial product that is being used in many sites including NASA, several National Laboratories and Universities.

1.1.3 System Architecture

A complete system architecture integrating on-line planning into a distributed, dual-arm robotic workcell has been developed. A careful analysis of the interfaces and world modelling issues has been performed, specifically concerning the interaction of planning with a hard real-time control system in the presence of unpredictable planning and communication delays. The final system incorporated a variety of subsystems:

- Human Interface. A graphical user interface containing a 3-dimensional rendering of the workspace allows the human to issue high-level task commands.
- Planner. A combined task-planner and path-planner[1] subsystem decomposes task commands into a sequence of safe, primitive "robot commands."
- Robot Workcell Controller and World Model. A hierarchical controller for the dual-arm workcell and the supporting world model. The world model integrates data from different sensors, taking advantage of domain-specific knowledge given the status of the workcell.
- Robot Workcell Simulator. A simulator capable of masquerading as the workcell while providing graphical simulation and world model functionality.
- Software Communication Bus. A Software package that allows transparent distributed network connectivity with unlimited fan-in and fan-out, thus providing the illusion of a software bus.

Strategic algorithms have been developed to allow tracking and capture of moving objects by one or two arms, as well as simultaneous capture of multiple objects (one per arm). These algorithms tie planned trajectories with local intercept trajectories and merge direct sensor-driven feedback (tracking) in some degrees of freedom with task-driven trajectories on other degrees of freedom. In addition, a scalable approach to the strategic control of a multi-arm workcell has been developed. This approach uses multiple instances of otherwise identical finite state machines to control each arm in the workcell.

A complete control hierarchy ranging from joint-level local torque control to high-level arm and object control has been demonstrated. Specifically, a set of mixed control modes for intercepting, tracking and picking moving objects smoothly from a conveyor has been constructed.

[1] These planners were developed at the Stanford Computer Science Robotics Laboratory.

The success of this d sign should provide a useful model for the development of similar systems.

1.1.4 System Interfaces

The selection of the system interfaces for the subsystems comprising an intelligent robotic system is often the most critical task, for these interfaces will determine the nature of the subsystems, the complexity of the system, its capabilities, and perhaps more importantly, how easy it is to modify and enhance.

As a result of this contract we have developed a design technique for complex robotic systems called interfaces-first design. Interfaces-first design develops information interfaces based on the characteristics of information flow in the system, and then builds subsystem interfaces from combinations of these information interfaces.

This technique is applied to a dual-arm workcell combining a graphical user interface, an on-line motion planner, real-time vision, and an on-line simulator. The system is capable of performing object acquisition from a moving conveyor belt and carrying out simple assemblies, without the benefit of pre-planned schedules nor mechanical fixturing.

- A philosophy for designing robotic systems beginning with the interfaces (*interfaces-first design*) has been proposed, developed, and demonstrated in a complete operational system. Furthermore, the concept of *anonymous* interfaces has been introduced and successfully demonstrated.
- A modular approach to developing system interfaces has been proposed and demonstrated. This technique structures the complex command and data flow as combinations of several *primitive* interface modules. These primitive interface modules have parameters that allow customization for use within a specific, larger interface.
- Three fundamental robotic-interface components: world-model, robot-command, and task-level-direction have been developed. These interfaces can be combined to generate custom interfaces between all the subsystems. A detailed study of the nature of the dataflow in robotic systems has led to the selection of these interfaces to encapsulate three fundamentally different classes of information:
  - World-State Interface. Encapsulates periodic, sensor-type information, where only the most current data is required. Different clients need this information at different rates to accommodate their specific needs.
  - Robot-Command Interface. Command-type information that must be delivered reliably and in order. This information is non-periodic and indicates an action that must be taken immediately or discarded.
  - Task-Interface. Goal-type information. Also requires reliable ordered delivery, and is generated asynchronously. However, it does not indicate an explicit action but rather a "success condition," and it persists until the goal is achieved or explicitly cancelled.

These interfaces alor : are capable of describing the system geometry and kinematics necessary for the planner and hun in-interface subsystems, providing a step toward the development of generic ("common") interfaces for robotic systems. They should serve as useful examples for similar systems.

1.1.5 Distributed Motion Planning

Motion planning is a relatively well understood problem, but it has been proven computationally hard. In order to allow motion planning techniques to be used in a real-time framework, we have developed the concepts of a new software architecture in which planning algorithms are distributed over multiple types of available resources. The goal of this architecture is to eventually provide the "best" on-line planning service to robot systems.

We have proceeded in three steps:

1. We have identified multiple axes along which motion planning computation can be usefully distributed. These axes are fairly general; they can apply to a wide variety of robotic problems.
2. We have conducted limited or substantial software experiments to validate each of the identified axes.
3. We have designed and implemented several planners, each one using a subset of the architectural concepts presented below.

The distribution axes that we have identified are the following:

- **Distribution over interconnected parallel processors:** It is the most obvious axis for distributing software. A given planning method may be parallelized so that it is concurrently executed on several processors, in order to get significant speedups. However, some planning methods may be more parallelizable than others. Furthermore, different sorts of parallelisms can be considered, e.g., coarse grain vs. fine grain.

  Software experiments: We have implemented two parallelized versions of RPP (the Randomized Path Planner that we developed under ARPA contract DAAA21-89-C0002): with two computer architectures: (1) a small-scale shared memory parallel multiprocessor machine and (2) a network of uniprocessor machines. Both implementations have brought significant speedup of the running time of RPP.

- **Distribution over problem approximations:** The most important parameters influencing the running time of a motion planner are well-known: the number of degrees of freedom, the dimension of the workspace, the number and the degree of the algebraic equations describing the objects' boundaries. Given a planning problem, the running time can often be reduced by one or several orders of magnitude by approximating the problem into a simplified, but still

realistic one, before running the planner. The approximation may, for example, consist of simplifying the shape of the objects and/or freezing some degrees of freedom of the robots. If the simplification is conservative, a solution of the simplified problem, if any, is also a solution of the original problem. However, there is no unique set of parameters which will work best in all situations. The tradeoff is between completeness and efficiency.

Software experiments: Both polyhedral and wire-frame models of robots have been made available to RPP. The wire-frame model approximates the robot as a set of cylinders. This approximation speeds up collision checking. The polyhedral model is available when the robot's shape is more complicated and cannot be reasonably approximated by cylinders. It provides more precise collision checking.

- **Distribution over planning methods:** Several practical motion planning methods have been developed over the last decade. Their operational characteristics (completeness, running time) are often different and there does not exist a single planner that is always better than all the others. In fact, several planners may be used to solve the same problem. A judicious choice among these planners for every planning problem may substantially improve efficiency. If several processors are available, one may try to run different planners concurrently to solve the same problem.

  Software experiments: Two modified versions of RPP have been implemented to explore the characteristics of different planning parameters in different pathological cases. These parameters include, in particular, the potential fields used to search the robots' configuration spaces. We have observed that different planning parameters have complementary advantages over the others, so that running them concurrently on several processors result in a significant reduction of the average running time.

- **Distribution of commitment over time:** A path is usually an overly committed motion plan to achieve a goal, since many alternative paths are equally acceptable. Committing a robot to following a given path has the drawback that any event may make the path invalid, requiring another path to be generated instead. Rather, the controller may do better by distributing commitment over time. For instance, in a first phase of processing, it may generate a "channel", i.e. a sort of tunnel containing a continuous infinity of paths for a robot; in a second phase, it may guide the robot through the channel using a simple potential field method.

  Software experiments: A landmark-based motion planner (see Subsection "Landmark-Based Mobile Robot Navigation" below) has been developed to generate reaction motion plans that anticipate uncertainties and allow decisions to be made at execution time based on the current sensed situations.

- **Distribution over time:** The controller should be opportunistic and distribute motion planning over time. Priority should be given to the most urgent computations (e.g., the generation of the motions to execute next), but, at any instant, if there are some extra free computing resources, these should be used to compute future motions according to what is

most likely to happen. Distribution of problems over time implies that the controller be able to decide to start an operation, even if it has not been fully planned.

Software experiments: Optimal-time motion planning techniques have being developed which transform a path into another path and its optimal command (see Subsection "Multi-Arm Manipulation Planning in 3D" below). The new path is locally optimal as far as the amount of time necessary to execute it is concerned. The temporal information generated by these techniques can later be used to decide how to distribute computation over time.

1.1.6 On-Line Manipulation Planning in Dynamic Environment

In the continuation of the work presented above, we have designed and implemented a on-line manipulation planning software for a dual-arm robot system constructing simple planar assemblies in a highly dynamic environment (moving parts, changing obstacles, changing goals). This planner distributes computation mainly along the time axis, i.e., it starts executing motion plans before these plans are complete. We believe that this planner is the first on-line sophisticated manipulation planner ever developed.

Our planner applies to the following type of robotic workcell. The workcell contains two SCARA-type manipulator arms. These arms must pick up parts arriving on a conveyer belt and deliver them at their specified goal locations, while avoiding collision with the obstacles in the environment. The parts arrive in random order at random time. The workspaces of the two arms intersect, so that the same part may be grasped by one arm, or the other (it is a planning issue to choose which one); also, an arm may grasp a part and later hand it over to the other arm if the goal is not reachable by the first arm. The obstacle locations and goal locations for the parts can be changed on-line at any time.

This scenario reflects the characteristics of the experimental testbed developed in ARL. The robotic workcell is typical of what is found in industry. However, in industry parts are constrained to arrive in fixed order and at given times, to ensure load balancing between the robot arms. This requires complex, long, and costly prior engineering. One of our goals was precisely to show that on-line planning has the potential to reduce the time and cost needed to setup a new robotic workcell. We have integrated our manipulation planner in a large software system under ControlShell. This system runs on the ARL experimental testbed (see Chapter 9).

Since almost everything in the scenario is dynamic and flexible, the needs for on-line planning is clearly motivated. Due to the complication of task-level planning and the huge search space it entails, providing a reasonably good on-line solution to this problem was extremely challenging. Our solution consists of orchestrating several planning primitives according to occuring events (e.g., arrival of new part, change in obstacle position). Each primitive solves a limited subproblem, but is extremely fast. The orchestration of the primitive is based on the decomposition of the total manipulation planning problem along several axes presented in the previous subsection.

We decompose the manipulation problems into several subproblems: catching a moving part on the

conveyor, delivering a grasped part to its goal, ungrasping and regrasping a part at an intermediate locations in order to avoid joint limits, etc. For each subproblem, we consider several specific cases: for example: an arm must grasp an arriving part, while the other arm is, or is not, moving. Each case is treated by a different planning primitive, which always search a low-dimensional configuration space (two or three dimensions). When a subproblem is solved, the generated plan is executed and the corresponding motion is treated as a new constraint to solve the next subproblems.

The primitives of the planner incorporate several new techniques. Their coordination is based on new ways to decompose and approximate problems. All these techniques are described in detail in Chapter 2.

We have conducted a considerable amount of experiments with the planner, both with a simulated testbed and the ARL experimental tesbed. The results, which are reported in Chapter 2, have been extremely satisfactory. We compared the time performance of the planner with a program used by Adept, Inc., to solve a similar, but more specific, problem. Although our planner is slightly slower, the running times of the two software modules are comparable. On the other hand, our planner did not benefit from, and does not require, prior engineering constraining the dynamic events occurring in the workcell. We believe that in many cases its greater generality and the increase in flexibility that it provides at runtime outweighs its slightly slower performance.

The main lesson from this planner is that complex on-line motion planning is now technically feasible, by appropriately orchestrating well-designed planning primitives. However, our planner is specific to a certain family of tasks, and it remains an open question whether this lesson is fully generalizable.

This planner has been the topic of Tsai-Yen Li's Ph.D. thesis (see Section 1.6).

1.1.7 Multi-Arm Manipulation Planning in 3D

The manipulation planner presented in the previous section exploits the specific kinematics of the SCARA robot arms to reduce the number of dimensions of the workspace to two and to break the manipulation problem into subproblems in configuration spaces having three dimensions at most (see Chapter 2). Concurrently, we have explored manipulation planning with an arbitrary number of robots having arbitrary kinematics and operating in 3D workspaces. This research led us to extend our Randomized Path Planner (RPP). We have also investigated path optimization techniques in this context. The manipulation planner developed is introduced below and described at greater length in Chapter 6.

Manipulation planning has been barely addressed so far in the motion planing literature. Existing publications consider extremely simplified versions of the problem. Often they have not led any implemented planners. The on-line manipulation planner of the previous section and the planner presented below are certainly the first realistic planners designed and implemented.

Our 3D multi-arm manipulation planner is a two-phase planner, which distributes computation

over problem approximations:

1. In the first phase, it mostly ignores the arms and uses RPP to plan the path of the object M to be moved from its initial to its goal configuration. It does so by invoking RPP to search the six-dimensional configuration space of M. The version of RPP used here has been slightly modified: it constructs a path τ of M so that M is "graspable" all along τ. We say that M is graspable, if the grasp points specified on M are reachable by available arms (see Chapter 6 for more detail).

2. In the second phase, the planner computes the path of the arms, trying to minimize the change of grasps. It selects the first grasp in order to maximize the subpath τ_1 of τ that can be performed without changing grasp. Then it selects the second grasp in the same way, and so on. The path of the arms alternate transit subpaths and transfer subpaths. The transfer subpaths move M along a subpath τ_i. They are entirely defined by the arms' inverse kinematics. The transit subpaths are generated by invoking RPP.

The planner allows the generation of manipulation plans where the object M can be moved by a single arm at a time, as well as manipulation plans where M must be moved by two (or more) arms.

The planner has been experimented in simulated environments, using different arrangements of robot arms (mostly workcells with two or three six-degree-of-freedom arms). Although it is not complete – in the sense that it may fail to find a solution path when one exists – it turned out to be quite reliable and also reasonably fast (complex paths, with multiple regraps, were generated in two to three minutes on a DEC Alpha computer).

We have investigated an extension of this manipulation planner aimed at producing manipulation trajectories (paths parameterized by time) that optimize a dynamic criterion (typically, minimize the duration of the motion). For that purpose, we used Bobrow's algorithm that we adapted to handle closed-loop chains (for the cases where the same object is moved by more than one robot at the same time). In the criterion to optimize, we also introduced the constraint that the moved object should not slip in the grasp. This constraint makes use of a simple friction model of the contact between the grippers and the part. This extension was implemented and tested in simulation. The temporal information generated by this extension can be used to decide how to distribute computation over time; but we have not explored this possibility.

This manipulation planner has been the topic of Yoshihito Koga's Ph.D. thesis (see Section 1.6).

1.1.8 Probabilistic Roadmap Planner

The RPP planner mentioned above has been successful in solving many complex planning problems involving many degrees of freedoms (i.e., configuration with many dimensions). However, we have identified some nasty pathological cases that it cannot solve in reasonable amount of time. We have

analyzed these cases, and this work led us to propose and design a totally new approach to path planning, based on the concept of probabilistic roadmaps. We implemented a planner (actually, several versions of this planner) based on this approach. We present this planner below. A more detailed description is given in Chapter 8. Some of the work related to this planner was done in cooperation with the group of Prof. Mark Overmars at Utrecht University (The Netherlands).

The probabilistic roadmap planner is based on distributing computations over two phases. In the first phase, preprocessing, the planner selects several configurations, called milestones, at random in the collision-free subset of the configuration space (the free space) and then tries to connect them in a network using a very simple and fast "local planner". This network is the generated roadmap. In the second phase, query processing, the planner is given a pair of configurations; it tries to find a path between them by connecting them to two milestones in the roadmap and checking that these milestones belong to the same component of the roadmap.

Both phases include special techniques to improve efficiency. For example, the preprocessing uses the degree of each milestone m in the roadmap, as it is constructed, to determine whether the milestone lies in a "difficult" region of the free space. The degree is the number of connections of m with other milestones found by the local planner. A small degree is a good indication that m lies in a difficult region. If this is the case, the planner randomly generates additional milestones in the vicinity of m. Hence, the final distribution of milestones over the free space is not uniform.

We conducted many experiments with the probabilistic roadmap planner, both in two-dimensional and three-dimensional workspaces, with a variety of simulated robots. These experiments show that the planner is extremely reliable (if enough milestones are generated in the preprocessing phase), that proprocessing is reasonably fast (from a few seconds to a few minutes, for most examples; but it may require several hours for really complicated problems), and that query processing is extremely fast (a small fraction of a second; often, less than 1/100 sec). Our experiments also demonstrate that the detection of milestones lying in difficult regions and the resampling operation in the vicinity of these milestones allow the planner to considerably reduce the number of milestones needed to generate roadmaps that represent well the connectivity of the free space.

In addition to these experiments, we formally analyzed the planning algorithm (actually, a variant of it). This analysis explains in mathematical terms the success of the implemented planner. It was published at the ACM Symposium on the Foundations of Computer Science (STOC) in 1995. We believe that this result is of major importance for future research. Between the complete planners that are totally impractical for more that 3 or 4 degree of freedom, and the adhoc planners that are too unreliable to be black boxes in larger systems, there exists a family of planners that are "probabilistically complete," i.e., the probability α that they fail to find a path when the running time t grows arbitrarily large converges toward zero. In the case of the probabilistic roadmap planner, our STOC paper shows that the function $t(1/\alpha)$ grows relatively slowly, which means that one can come very close to full completeness with reasonable computational time. We have expanded on this idea in a paper presented at the 7th International Symposium on Robotics Research (ISRR) in October 1995.

This probabilistic roadmap planner has been the topic of Lydia Kavraki's Ph.D. thesis (see Sec-

tion 1.6).

1.1.9 Landmark-Based Mobile Robot Navigation

Uncertainties may often cause robots to fail, thus undermining their reliability. One approach to dealing with this problem is to model uncertainties and develop reaction plans that can successfully face contingencies. However, in the past, attempts to reason about uncertainties at planning time have resulted in exponential-time algorithms. The applicability of these algorithms is limited to very simple worlds. Our research proposed an alternative approach: Investigate the limits of polynomiality, that is, find the most complex robot-workspace model that accepts a polynomial-time planner. This approach led the design of a polynomial planning algorithm for mobile robots with uncertainty in sensing and control. We have implemented this planner and experimented with it both in simulation and with real mobile robots. In addition to being quite efficient, the planner demonstrated its ability to generate reliable plans.

We define uncertainty in sensing (or control) as the range of possible errors in sensing (or control). We assume this range to be bounded. A guaranteed plan is then a plan whose execution is guaranteed to succeed in achieving the goal, for any combination of errors within their ranges. Bounding error ranges often requires engineering the workspace and/or the robot appropriately. We believe that this cannot be avoided. Even if one drops a robot on Mars, this robot will be equipped with sensors that are appropriate to deal with the features of Mars environment. Our approach to reliable robot navigation thus consists of bounding errors through appropriate engineering and running a planner that generates guaranteed plans. We are interested in minimizing engineering, as long as the planning problem remains tractable (i.e., polynomial). Our planning algorithm achieves polynomiality by introducing a formal concept of landmark. In addition to generating guaranteed plans, it can also be used to generate probabilistic plans (plans that may fail, but which minimize the probability of failure). This planner is described at length in Chapter 7.

We assume that the robot is a point in a two-dimensional configuration space (this corresponds, for example, to the case where the robot is a circular omni-directional mobile robot). This space consists of three types of regions: obstacles, landmarks, and the rest. The robot is not allowed to enter any obstacle. Whenever it enters a landmark region L, it sensors are guaranteed to detect that the robot is in L. Moreover L contains one or several "kernels" $K_1, \ldots, K_n$, such that from anywhere in L the robot can reliably attain any selected kernel. In the rest of the space, the robot does not know where it is (i.e., sensing uncertainty is infinite) and if it tries to move along a certain direction d, it actually moves along any direction $d \pm \theta$, where θ is the control uncertainty. In Chapter 7 we discuss the adequacy of this model. We also present a simple effective landmark design that corresponds to this model. This is the landmark design with which we have experimented. We now routinely use this type of landmark.

The planner works backward from the goal region it must achieve, by computing the omni-directional preimage of a set of an iteratively growing set of landmark regions. The directional preimage of a set S of landmarks, for a commanded direction of motion d, is the maximal region

from which the robot can start a motion along any direction $d \pm \theta$ and still be guaranteed to reach a landmark in S. The omni-directional preimage of S is the set of all the preimages, for all the directions d, indexed by these directions. Our planner relies on the fact that one can fully represent this set by considering only a polynomial number of directional preimages. The planner first computes the omni-directional preimage of the landmark regions intersecting the goal. In any directional preimage in it contains the initial region of the robot, then the planning problem is solved. Otherwise, if a directional preimage intersects a new landmark region, this region is inserted in S. Etc. The planner fails if at some iteration no directional preimage contains the initial region or intersect a new landmark region.

Several extensions of the planners have been investigated and implemented: dealing with variable control uncertainty, dealing with unexpected obstacles (obstacles that were not part of the planner's model), and generation of probabilistic plans.

This landmark-based planner has been the topic of Anthony Lazanas' Ph.D. thesis (see Section 1.6).

1.1.10 Mobile Robot Navigation Toolkits

Developing a reliable, integrated mobile robot software system from scratch is a time-consuming task. This is because a mobile robot navigation system consists of various interacting components. However, many of these components are basic and common among systems. Therefore, there is no need for "re-inventing the wheel," whenever a system is implemented to satisfy the needs of a new application. The key idea therefore is to develop and maintain reusable software modules, which we call toolkits. These toolkits are used as tools and building blocks for constructing various mobile robot experimentation systems.

We have designed a layered set of toolkits:

1. The Geometry Engine Layer: It provides the common geometric functions required by the various toolkits in the Building Blocks Layer, e.g.: line and curve intersection, polygon intersection and clipping.

2. The Building Blocks Layer: It consists of various toolkits that support the development of the different components of a mobile robot software system:

   - Robot Modeling Toolkit.
   - World Modeling Toolkit.
   - Path Tracking Toolkit.
   - Sensor-Based Localization Toolkit.
   - Map Building Toolkit.
   - Motion Planning Toolkit.

3. The System Developer Layer: It consists of various structures to help organize the components from the different toolkits into a reliable navigation system.

4. The Simulator Layer: Although experimenting with real robots is needed to conduct realistic research with mobile robots, it often turns into a frustrating and painful experience. To overcome these difficulties, we have developed a graphic multi-robot simulator (with the robots' sensors). Robot programs can alternatively run with this simulator or with real robots.

These toolkits have been transferred to Nomadic technologies (some of them were developed jointly). Several of them are now commercially available. A large portion of the experimental software that we now develop for mobiles robots reuse a subset of these toolkits.

1.2 Applications and Technology Transfer

It is not possible to develop generic technology without multiple, specific applications to test and refine the ideas and implementations. As such, we are actively seeking sites, both internally and externally to provide the compelling test beds that will make this project succeed. These driving applications span a variety of the most important target users: high-performance control, intelligent machine systems, underwater vehicle command and control, and remote teleoperation. Several of these projects will reach for new limits in advanced technology and system integration; others will address real-world problems in operational systems.

With the reduced funding levels, we did not have the resources to support all of the originally proposed technology evaluation sites. However, we believe these sites are crucial to the development of ControlShell into a viable technology for "real-world" use. Thus, we have actively pursued alternative means of supporting external sites. We have been successful in securing several new test applications. These sites will either function with minimal support, or fund their own support.

This chapter highlights some of the activities of these projects.

The currently-active ControlShell applications are:

- Precision Machining, by The Stanford Quiet Hydraulics Laboratory.
- Underwater Vehicle Control, a joint project between the ARL and the Monterey Bay Aquarium Research Institute.
- Intelligent Machine Architectures, by Lockheed Missiles and Space Corporation.
- Remote Teleoperation, by Space Systems Loral Corporation.
- Space-based Mobile Robot Systems, by several ARL students (NASA-sponsored).

- High-Performance Control of Flexible Structures, by several ARL students (AFOSR-sponsored).
- Space-structure assembly, by NASA Langley Research Center.
- Mobile-robot control by NASA Ames Research Center.
- Marsokhod Martian Rover, by NASA Ames Research Center.
- Robotic Cleanup, by Oak-Ridge National Laboratories.
- Space Robotic Teleoperation, by NASA Jet Propulsion Laboratory.
- Machining, by the National Institute of Standards and Technology.
- Robotics for Hazardous environments, by Oak-Ridge National Laboratories.
- Robotics for Radioactive-Materials Handling, by Lawrence Livermore National Laboratories.

1.3 NASA sites

Three NASA sites have committed to using ControlShell in their projects. The first site is a robotics laboratory at NASA's Langley Research Center. This laboratory will be studying robotic construction of space structures, using a singleindustrial robot. The work focuses on end-effector development, system integration, and tools to ease construction planning.

Another NASA lab, the intelligent mechanisms group at NASA Ames [2], is also using ControlShell for a mobile robotics project. This project's goals include studying intelligent exploration algorithms and robotic control architectures.

The third NASA lab, the telerobotics group at JPL is using ControlShell in several projects, including a lunar rover [3], an a vehicle to be used in Venus.

1.4 MBARI Underwater Vehicle

The Monterey Bay Aquarium Research Institute (MBARI), in a joint program with ARL, is using ControlShell to develop the control system of the OTTER (Ocean Technology Testbed for Engineering Research) vehicle [4]. The vehicle is a unmanned submersible about 2 meters long and weighing 150 kilograms. It is powered by 8 thrusters and has an active vision sensing system. The vehicle will be used in research focusing on semi-autonomous control. On the vehicle, ControlShell is the the backbone of the control software that runs the thrusters and processes the sensor inputs.

[2] http://maas-neotek.arc.nasa.gov/Marsokhod/marsokhod.html
[3] http://robotics.jpl.nasa.gov/tasks/rovertech/homepage.html
[4] http://sun-valley.tanford.edu/projects/underwater_robotic_vehicles/underwater_robotic_vehicles.html

Control laws and data processing filters implemented in ControlShell allows us to feedback sensor input for automatic positioning and station-keeping.

The Finite-State Machine facility of ControlShell is used at the supervisory level to coordinate vehicle actions in a programmed response to stimuli. Stimuli can be produced by automatic sensing and error detection processes or through direct user interaction. FSM's are used to program complex vehicle actions to complete tasks autonomously.

The project uses NDDS for all interprocess/interprocessor communications. The two independent real-time computing systems on-board the vehicle are connected to a off-board real-time computer through a SLIP (Serial-Level Internet Protocol) line during operations. That provides network connectivity—albeit at a slower rate—even with the slow underwater acoustic modem communications. In addition, the graphical user interface running on an HP workstation is connected via Ethernet to the off-board real-time system. NDDS provides the basic communications and message passing between all four independent systems. Since NDDS is network transparent, any of the computing systems can be replaced with a simulation during development to test software and other components of the entire OTTER system.

1.5 Transfer of Planning Technology

- Organization of a one-day workshop on Assembly Planning and Concurrent Design at Stanford in November 1992, with representatives from the following companies: Hewlett-Packard, IBM, Ford, Boeing Commercial Airplane Group, Boeing Computer Services, Silma, Sapphire Design Systems, Adept Technolgies.
- One Ph.D. student, Bruce Romney, spent the summer '93 at the GM Research Labs, in Warren (MI), to investigate the application of our manipulation and assembly planning techniques to industrial manufacturing. Several of our planners have been made available to GM. We are continuing our interaction with them.
- One Ph.D. student, Tsai-Yen Li, spent the summer '94 at the GE Research Labs, in Schenectady (NY). There, he integrated one of our path planners with a GE graphic system to help planning for maintenance operation on aircraft engines. The planner is now routinely used by GE personnel, who claimed that the planner resulted in a 500% gain in planing efficiency.
- We have assisted Nova Management, Inc., in building an automated route planner for a tank simulator in support of US Government Contract No. DAAE07-C-93-0026.
- Two of our planners, RPP and the probabilistic planners, have been made available through anonymous ftp at `flamingo.stanford.edu:/pub/li/rpp3d.tar.gz` (for RPP) and `flamingo.stanford.edu:/pub/kavraki/prm.tar.gz` (for the probabilistic roadmap planner). Several research institutions and companies have imported these planners.
- We have adapted our randomized motion planning techniques for a new radio-surgery system developed jointly by the Stanford Medical Center and Accuray, Inc. This radio-surgery system

consists of a general GMF robot arm moving a light-weight linear accelerator and an X-ray vision system that locates the patient's skull in real-time. The planner computes the sequence of directions from which the linear accelerator should shoot at the brain tumor, without damaging healthy tissue. It also generates the actual path of the robot to generate this sequence of directions. Several patients have been operated (brain and spinal cord tumors).

- We have transferred our software toolkits for mobile robots to Nomadic technologies, Inc., who has converted them into a commercial product sold with their robots. Nomadic is a leading provider of mobile robots for research. It has sold more than 150 robots worldwide.
- We have made several presentations of our work to companies, like Boeing, IBM, GE, GM, SUN Microsystems, Alcoa, Adept, etc., as well as to military and DOE labs (Army Research Laboratory in Aberdeen, MD; Sandia National Labs in Albuquerque, NM).

1.6 Ph.D. Students

The following Ph.D. students, who did their research as part of this project, have graduated:

- Lydia Kavraki, Ph.D. in Computer Science, Thesis title: *Random networks in Configuration Space for Fast Path Planning.*

  Current position: Research Associate, CS Dept., Stanford University.

- Yoshihito Koga, Ph.D. in Mechanical Engineering, Thesis title: *On Computing Multi-Arm Manipulation Trajectories.*

  Current position: CTO, The Motion Factory, Inc., Cupertino, CA.

- Anthony Lazanas, Ph.D. in Computer Science, Thesis title: *Reasoning About Uncertainty in Robot Motion Planning.*

  Current position: Lehman Brothers, New York.

- Tsai-Yen Li, Ph.D. in Mechanical Engineering, Thesis title: *On-Line Motion Planning in a Dynamic Environment.*

  Current position: Assistant Professor, Department of Computer Science, National Chengchi University, Wenshan, Taipei, Taiwan.

- Gerardo Pardo-Castellote, Ph.D. in Electrical Engineering, Thesis title: *Experiments in the Intagration and Control of an Intelligent Manufacturing Workcell.*

  Current position: Principal, Real-Time Innovations, Inc., Sunnyvale, CA.

- Larry Pfeffer, Ph.D. in Mechanical Engineering, Thesis title: *The Design and Control of a Two-Armed, Copperating, Flexible-Drivetrain Robot System.*

  Current position: National Institute of Standards and Technology, Gaithersburg, MD.

Publications So Far:

J. Barraquand, L. Kavraki, J.C. Latombe, T.Y. Li, R. Motwani, and P. Raghavan, A Random Sampling Scheme for Path Planning, to appear in *Proc. 7th Int. Symp. on Robotics Research*, G. Giralt and G. Hirzinger (eds.), Herrsching, Germany, October 1995.

R.I. Brafman, J.C. Latombe, and Y. Shoham, "Towards Knowledge-Level Analysis of Motion Planning," *Proc. of the 11th Nat. Conf. on Artificial Intelligence*, AAAI-93, Washington D.C., July 1993. pp. 670-675.

R.I. Brafman, J.C. Latombe, Y. Moses, and Y. Shoham, "Knowledge as a Tool in Motion Planning Under Uncertainty," *Proc. of the 5th Conf. on Theoretical Aspects of Reasoning About Knowledge*, Monterey, CA, Morgan-Kaufmann Publishers, 1994, pp. 208-224.

H. Chang and T.Y. Li, Assembly Maintainability Study with Motion Planning, *Proc. IEEE Int. Conf. on Robotics and Automation*, Nagoya, Japan, 1995, pp. 721-728.

L. Kavraki, *Computation of Configuration-Space Obstacles Using the Fast Fourier Transform*, Technical Report, STAN-CS-92-1425, 1992.

L. Kavraki, *Computation of Configuration-Space Obstacles Using the Fast Fourier Transform*, *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Atlanta, GA, 1993.

L. Kavraki, J.C. Latombe, and R.H. Wilson, "On the Complexity of Assembly Partitioning," *Proc. of 5th Canadian Conf. on Computational Geometry*, August 1993, pp. 12-17.

L. Kavraki, J.C. Latombe, and R.H. Wilson, On the Complexity of Assembly Partitioning, *Information Processing Letters*, 48, 1993, pp. 229-235.

L. Kavraki and J.C. Latombe, *Randomized Preprocessing of Configuration Space for Fast Path Planning*, Rep. No. STAN-CS-93-1490, Dept. of Computer Science, Stanford University, September 1993.

L. Kavraki and J.C. Latombe, *Randomized Preprocessing of Configuration Space for Fast Path Planning*, *Proc. of the IEEE Int. Conf. on Robotics and Automation*, San Diego, May 1994, pp. 2138-2145.

L. Kavraki, P. Svestka, J.C. Latombe, and M. Overmars, Probabilistic Roadmaps for Path Planning

in High-Dimensional Configuration Spaces, accepted for publication in *IEEE Tr. on Robotics and Automation.* (also published as Tech. Rep. STAN-CS-TR-94-1519, Dept. of Computer Science, Stanford University, August 1994.)

L. Kavraki and J.C. Latombe, *Randomized Preprocessing of Configuration Space for Path Planning: Articulated Robots, Proc. of the IEEE Int. Conf. on Intelligent Robots and Systems*, Munich, Germany, September 1994.

L. Kavraki, J.C. Latombe, R. Motwani, and P. Raghavan, *Randomized Query Processing in Robot Motion Planning*, accepted for publication in *Proc. of ACM Symp. on Theory of Computer Science (STOCS)*, 1995.

L. Kavraki, Computation of Configuration-Space Obstacles Using the Fast Fourier Transform. *IEEE Tr. on Robotics and Automation*, 11(3), 1995, pp. 408-413.

L. Kavraki, J.C. Latombe, R. Motwani, and P. Raghavan, Randomized Query Processing in Robot Motion Planning, *Proc. of ACM SIGACT Symp. on Theory of Computer Science (STOC)*, Las Vegas, Nevada, 1995, pp. 353-362.

Y. Koga and J.C. Latombe, "Experiments in Dual-Arm Manipulation Planning," *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Nice, May 1992, pp. 2238–2245.

Y. Koga, T. Lastennet, J.C. Latombe, and T.Y. Li, "Multi-Arm Manipulation Planning," *Proc. of the 9th Int. Symp. on Automation and Robotics in Construction*, Tokyo, June 1992.

Y. Koga and J.C. Latombe, "On Multi-Arm Manipulation Planning," *Proc. of the IEEE Int. Conf. on Robotics and Automation*, San Diego, May 1994, pp. 945–952.

Y. Koga, K. Kondo, J. Kuffner, and J.C. Latombe, "Planning Motions with Intentions," *Proc. of SIGGRAPH'94*, July 1994, pp. 395-408.

J.C. Latombe, "Geometry and Search in Motion Planning," *Annals of Mathematics and Artificial Intelligence*, 8(2-4), 1993.

J.C. Latombe, "Robot Algorithms," *Proc. of the LAAS/CNRS 25th Anniversary Conf.*, Cepadues, Toulouse, France, May 1993, pp. 81–94 (invited conference).

J.C. Latombe, "Robot Algorithms," presented at the WAFR Workshop held in San Francisco, CA, February 1994. To appear in*Foundations of Robot Algorithms*, D. Halperin, K. Goldberg, J.C. Latombe, and R.H. Wilson (eds.), AK Peters, Wellesley, MA, 1995.

J.C. Latombe, "Robot Algorithms," *Proc. of the 6th Intl. Symp. on Robotics Research*, T. Kanade and R. Paul (eds.) The International Foundation for Robotics Research, Cambridge, MA, 1994, pp. 5-20.

J.C. Latombe, Robot Algorithms, *Foundations of Robot Algorithms*, K. Goldberg, D. Halperin, J.C. Latombe, and R.H. Wilson (eds.), AK Peters, Wellesley, MA, 1995, pp. 1-18.

A. Lazanas and J.C. Latombe, *Landmark-Based Robot Navigation*, Rep. No. STAN-CS-92-1428,

Dept. of Computer Science, Stanford U., May 1992. Accepted for publication in *Algorithmica.*

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Navigation," *Proc. of the 10th Nat. Conf. on Artificial Intelligence, AAAI-92*, San Jose, July 1992, pp. 816–822.

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Motion Planning," *Proc. of the AAAI Fall Symp.*, Boston, MA. October 1992, pp. 98-103.

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Motion Planning," *Geometric Reasoning for Perception and Action*, C. Laugier (Ed.), Lecture Notes in Computer Science, 708, Springer-Verlag, 1993.

A. Lazanas and J.C. Latombe, Landmark-Based Robot Navigation, *Proc. of 10th Nat. Conf. on Artificial Intelligence (AAAI-92)*, San Jose, July 1992, pp. 816-822.

A. Lazanas and J.C. Latombe, Landmark-Based Robot Navigation, *Algorithmica*, 13, 1995, pp. 472-501.

A. Lazanas and J.C. Latombe, Motion Planning with Uncertainty: A Landmark Approach, *Artificial Intelligence*, 76(1-2), 1995, pp. 285-317.

T.Y. Li and J.C. Latombe, "On-Line Motion Planning for Two Robot Arms in a Dynamic Environment," accepted for publication in *Proc. of the IEEE Intl. Conf. on Robotics and Automation*, Nagoya, Japan, May 1995.

T.Y. Li and H. Chang, Design for Maintenance by Constrained Motion Planning, *Proc. ASME Design Engineering Tech Conf.* Vol. 2, 1995, pp. 869-876.

T.Y. Li and J.C. Latombe, On-Line Motion Planning for Two Robot Arms in a Dynamic Environment, accepted for publication in *The Int. J. of Robotics Research.*

Gerardo Pardo-Castellote and Robert H. Cannon Jr. "Proximate time-optimal parameterization of robot paths," STAN-ARL-92- 88, Stanford University Aerospace Robotics Laboratory, April 1993.

Gerardo Pardo-Castellote, Tsai-Yen Li, Yoshihito Koga, Robert H. Cannon Jr., Jean-Claude Latombe, and Stan Schneider," "Experimental integration of planning in a distributed control system. In *Preprints of the Third International Symposium on Experimantal Robotics*, Kyoto Japan, October 1993.

Gerardo Pardo-Castellote and Stanley A. Schneider. "The network data delivery service: real-time data connectivity for distributed control applications," *Proceedings of the International Conference on Robotics and Automation*, San Diego, CA, May 1994. IEEE, IEEE Computer Society.

Gerardo Pardo-Castellote and Stanley A. Schneider. "The network data delivery service: A real-time data connectivity system," In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space*, Houston, TX, March 1994. AIAA, AIAA.

Gerardo Pardo-Castellote, Stanley A. Schneider and R. H. Cannon Jr. "System Design and Interfaces for Intelligent Manufacturing Workcell," *Proceedings of the International Conference on*

Robotics and Automation, Nagoya, Japan, May 1995. IEEE, IEEE Computer Society.

S. Schneider and R. H. Cannon. "Object impedance control for cooperative manipulation: Theory and experimental results," *IEEE Journal of Robotics and Automation*, 8(3), June 1992. Paper number B90145.

Stanley A. Schneider Vince Chen and Gerardo Pardo-Castellote. "ControlShell: A Real-Time Software Framework," In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space*, Houston, TX, March 1994. AIAA, AIAA.

S. A. Schneider and R. H. Cannon. "Experimental object-level strategic control with cooperating manipulators," *The International Journal of Robotics Research*, 12(4):338–350, August 1993.

Stanley A. Schneider, Vince W. Chen and Gerardo Pardo-Castellote. "The ControlShell Component-Based Real-Time Programming System," *Proceedings of the International Conference on Robotics and Automation*, Nagoya, Japan, May 1995. IEEE, IEEE Computer Society.

Stanley A. Schneider, Vince W. Chen, Jay Steele and Gerardo Pardo-Castellote. "The ControlShell Component-Based Real-Time Programming System, and its Application to the Marsokhod Martian Rover" *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 1995. ACM.

Howard H. Wang, Richard L. Marks, Stephen M. Rock, and Michael J. Lee. "Task-based control architecture for an untethered, unmanned submersible," In *Proceedings of the 8th Annual Symposium of Unmanned Untethered Submersible Technology*, pages 137–147. Marine Systems Engineering Laboratory, Northeastern University, September 1993.

R.H. Wilson, L. Kavraki, J.C. Latombe, and T. Lozano-Pérez, Two-Handed Assembly Sequencing, *The Int. J. of Robotics Research*, 14(4), 1995, pp. 335-350.

Chapter 2

ControlShell: A Real-Time Software Framework

This chapter is based on the paper "ControlShell: A Real-Time Software Framework," by Stan Schneider, V. Chen, and G. Pardo-Castellote, published in Proc. of IEEE Int. Conf. on Robotics and Automation, *Nagoya, Japan, May 1995*

2.1 Introduction

Motivation System programs for real-time command and control are, for the most part, custom software. Emerging operating systems [28, 52, 60, 50, 40] provide some basic building blocks—scheduling, communication, etc.—but do not encourage or enable any structure on the application software. Information binding and flow control, event responses, sampled-data interfaces, network connectivity, user interfaces, etc. are all left to the programmer. As a result, each real-time system rapidly becomes a custom software implementation. With so many unique interfaces, even simple modules cannot be shared or reused.

An effective real-time framework must create a programming environment that facilitates sharing and reuse of real-time program modules. At a minimum, this requires providing interface specifications and data transfer mechanisms. The framework must also provide services and tools to combine modules and build systems from reusable components. Finally, the framework must meet the many challenges unique to real-time computing. For example:

- Real-time code must be able to react to external temporal events.
- The real-time execution environment is *fundamentally* multi-threaded and asynchronous.

- Real-time systems are usually composed of several different layers of control, each with different characteristics. For instance, strategic-level command and low-level servo control must be blended into a smoothly-operating system.
- Real-time systems must handle changing conditions, often requiring switching between drastically different modes of operation.
- Real-time systems are often physically distributed. In the simplest case, an operator control station may be remotely situated. More complex systems are comprised of many interacting distributed real-time and non-real-time subsystems.

All these challenges must be efficiently and smoothly handled by the architecture.

2.1.1 ControlShell's Solutions

Component-Based Design ControlShell is specifically designed to address these issues. ControlShell provides interface definitions and mechanisms for building real-time code modules. ControlShell also provides basic data structure specifications, and mechanisms for binding data with routines and specifying data-flow requirements. These two critical features make simple generic packages (known as *components*) possible. ControlShell systems are built from combinations of these components.

An extensive library of pre-defined components is provided with the system, ranging from simple filters and controllers to complex trajectory generators and motion planning modules. New or custom components are easily added to the system via the graphical *Component Editor* (CE). The Component Editor allows simple specification of data interchange requirements. Code is automatically generated to permit instancing the new component into the system.

Graphical CASE System-Building Tools ControlShell also provides a set of powerful development tools for building complex systems. Building a system is accomplished by connecting components within a graphical *Data Flow Editor* (DFE). The data flow editor resolves the system data dependencies and orders the component modules for most efficient execution. Radical mode changes are supported via a "configuration manager" that permits quick reconfiguration of large numbers of active component routines.

Real-time systems also require higher-level control functions. ControlShell's event-driven finite state machine (FSM) capability provides easy strategic control. The state machine model features rule-based transition conditions, true callable sub-chain hierarchies, task synchronization and event management. A graphical FSM editor facilitates building state programs.

Real-Time System Services To provide support for real-time distributed systems, ControlShell includes a network connectivity package known as the *Network Data Delivery Service* (NDDS).

NDDS provides distributed data flow. It naturally supports multiple anonymous data consumers and producers, arbitrary data types, and on-line reconfiguration and error recovery.

ControlShell also offers a database facility, direct support for sampled-data systems, a full matrix package, and an interactive menu system. Figure 2.1 presents an overview of the ControlShell toolset and design approach.

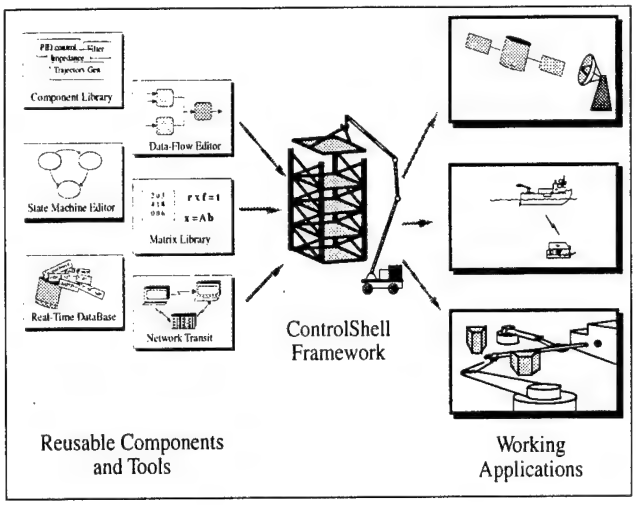


Figure 2.1: **ControlShell Design Process**

The ControlShell system designer uses the many powerful tools, system services, and prebuilt library components to construct a modular system.

2.2 Relation to Other Research

There are two quite different issues in real-time software system design:

- Hierarchy (what is communicated)
- Superstructure architecture (how it is communicated)

Several efforts are underway to define hierarchy specifications; NASREM is a notable example [2]. ControlShell makes no attempt to define hierarchical interfaces, but rather strives to provide a sufficiently generic software platform to allow the exploration of these issues. As such, this work takes a first step—defining the architecture superstructure (control and data flow mechanisms).

Several distributed data-flow architectures have been developed, including CMU's TCA/X [48, 11], Rice University's TelRIP [61], and Sparta's ARTSE [51]. These provide various levels of network

services, but do not address repetitive service issues or resolve multiple data-producer conflicts in a symmetric robust "stateless" architecture as does the ControlShell NDDS system (see [162] for details). Also, they are not integrated within a general programming system.

Recently, more sophisticated programming environments have begun to emerge. For example, ORCAD [49] and COTS [14] are specialized robotics programming environments. Two commercial products, System Build with AutoCode from Integrated Systems, Inc. [17], and SIMULINK with C-Code Generation from the MathWorks, Inc. [24] are sophisticated control development environments. They offer easy-to-use rapid control system prototyping. They are not, however, architectures well suited to developing complex multi-layer distributed control hierarchies.

Implementation Experience ControlShell evolved from many years of research with real-time control systems. It was first developed for use with a multiple-arm cooperative robot project at Stanford University's Aerospace Robotics Laboratory[43, 44, 45]. From this start, ControlShell spread to become the basis for more than 20 research projects in advanced control systems at Stanford. Among these were projects to study the control of flexible structures, adaptive control, control of mobile robots (including multiple coordinated robots), and high-bandwidth force control [20, 30, 55, 56, 7, 33]. More recently, a few industrial sites and two NASA centers have begun experimenting with ControlShell applications [53, 54]. ControlShell is now being jointly developed by Stanford University and Real-Time Innovations, Inc. It is supported under ARPA's Domain-Specific Software Architectures (DSSA) program.

This *continuous migration* from specific, working applications to wider spectrums of use is the key to usable generality. These applications continue to drive ControlShell's growth. To our knowledge, ControlShell is the only integrated framework package combining transparent networking, component-based system description, a state machine model, and a run-time executive.

2.3 Run-Time Structure

Some of the major system modules are shown in Figure 2.2. As shown in the figure, ControlShell is an *open* system, with application-accessible interfaces at each level. The figure is organized (loosely) into data and execution hierarchies.

At the lowest layer, ControlShell executes within the VxWorks real-time operating system environment. The simple base class known as *CSModules* is the building block for most executable constructs. Organizations of these modules, into lists, menus, and finite state machines form the core executable constructs. Users build useful execution-level atomic objects called *components* by defining derived classes from CSModules and binding them through the on-line data base to data matrices from the *CSMat* package. High-level graphical editors speed component definition, data flow specification and state machine programming. Network connectivity is provided by NDDS for all application modules.

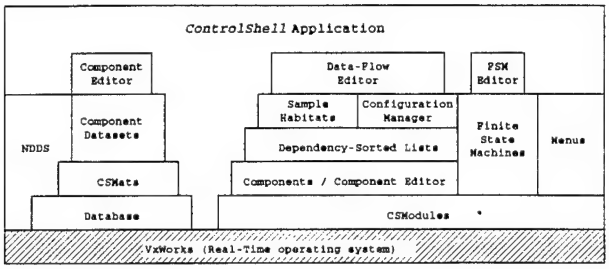


Figure 2.2: **Run-Time Structure**

ControlShell's open architecture provides many powerful services, while allowing application code free access to the underlying structures.

2.4 Data Flow Design

Many real-time systems contain sampled-data subsystems. Here, we define a "sampled-data" system as any system with a clearly periodic nature. Common examples (each of which have been implemented under ControlShell) are digital control systems, real-time video image processing systems, and data acquisition systems. Each of these is characterized by a regular clock source.

Providing an environment where sampled-data program components can be interchanged is challenging. These programs have routines that must be executed during the sampling process, routines to initialize data structures (or hardware) when sampling begins, and perhaps to clean up when sampling ends. Further, many routines are dependent on knowledge of the timing parameters, etc. Although they may interact—say by passing data—sampled-data program components are often relatively independent. Requiring the application code to call each module's various routines directly destroys modularity.

2.4.1 Components

The *component* is the fundamental unit of reusable data-flow code in ControlShell. Components consist of one or more *sample modules* derived from CSModules. Sample modules have several pre-defined entry routines, including:

Routine	When executed
execute	Once each sample period
stateUpdate	After all executes are done
enable	When this module is made active
disable	When it is removed from the active list
startup	When sampling begins
shutdown	When sampling ends
timingChanged	When the sample rate changes
reset	When the user types "reset", or calls CSSampleReset
terminate	When the module is unloaded

Thus, a motor driver component might define a startup routine to initialize the hardware, an execute routine to control the motor, and a shutdown routine to disable the motors if sampling is interrupted for any reason. In addition, if any of its parameters depend on the sampling rate, it may request notification via a timingChanged method. By allowing components to attach easily to these critical times in the system, ControlShell defines an interface sufficient for installing (and therefore sharing) generic sampled-data programs.

Building Components: The Component Editor An easy-to-use graphical tool called the *component editor* (CE) assists the user in generating new components and specifying their data-flow interactions. The component editor defines all the input and output data requirements for the component, and creates a data type for the system to use when interacting with the component. The tool contains a code generator; it automatically generates a description of the component that the Data-Flow Editor can display (see below), and the code required to install instances of the component into ControlShell's run-time environment.

2.4.2 Execution Lists

An execution list is simply a dynamically changeable, ordered list of sample modules to be sequentially executed. The active set of modules on a list can be changed anytime. In fact, lists may drastically change their contents during system mode changes.

Execution lists may be sorted to provide automated run-time execution scheduling to resolve data dependencies. More specifically, the modules are sorted so that data consumers are always preceded by the appropriate data producers (see Figure 2.3). The system uses the specifications of the data flow requirements for each component to sort the dependencies and order the list. A side benefit of the sorting process is the error-checking that is performed to insure consistent data flow patterns.

Sample Habitats ControlShell provides a named sampled-data environment, known as a *sample habitat.* A sample habitat encapsulates all the information and defines all the interfaces required

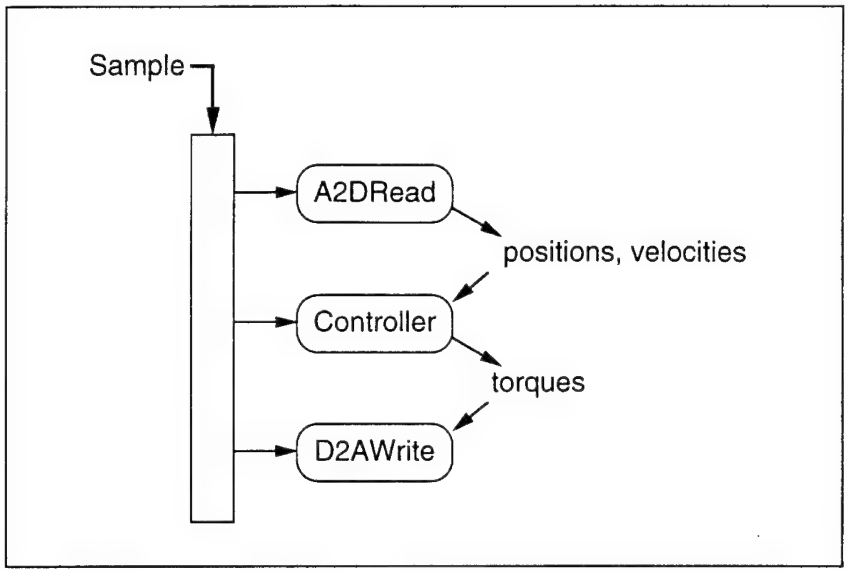


Figure 2.3: **Dependency-Sorted List**

Dependency-sorted execution lists provide automatic run-time sorting by data *dependencies.*

for sampled-data programs to co-exist. It also contains routines to control the sampling process. For example, a module installed into a sample habitat can query its clock source and sample rate, start and stop the sampling process, etc.

Each sample habitat contains an independent task that executes the sample code. The task is clocked by the periodic source (such as a timer interrupt). Special components are provided to interface between habitats, allowing multi-rate controller designs.

2.4.3 Building Systems: The DFE Editor

Building systems of components is made simple by the graphical Data-Flow Editor (DFE). The DFE reads description files produced by the component editor, and then allows the user to connect components in a friendly graphical environment. It allows specification of all the data connections in the system, as well as reference inputs—gains, configuration constants and other parameters to the individual components. An example session is depicted in Figure 2.4.

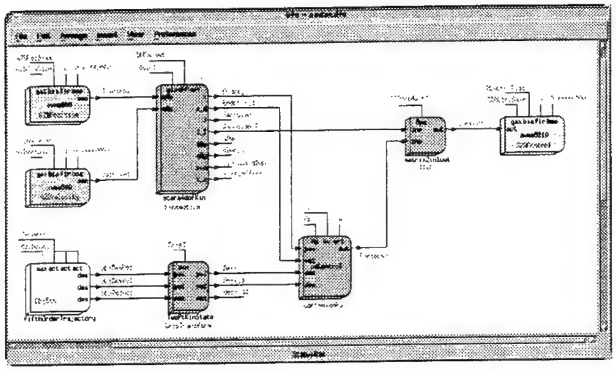


Figure 2.4: **Data-Flow Editor**

The data-flow editor builds collections of components into an executing system.

2.5 Configuration Management

Complex real-time systems often have to operate under many different conditions. The changing sets of conditions may require drastic changes in execution patterns. For example, a robotic system coming into contact with a hard surface may have to switch in a force control algorithm, along with its attendant sensor set, estimators, trajectory control routines, etc.

ControlShell's configuration manager directly supports this type of radical behavior change; it allows entire groups of modules to be quickly exchanged. Thus, different system personalities can be easily interchanged during execution. This is a great boon during development, when an application programmer may wish, for example, to quickly compare controllers (See Figure 2.5). It is also of great utility in producing a multi-mode system design. By activating these changes from the state-machine facility (see below), the system is able to handle easily external events that cause major changes in system behavior.

Configuration Hierarchy The configuration manager essentially creates a four-level hierarchy of module groupings. Individual sample modules form the lowest level. These usually implement a single well-defined function. Sets of modules, called *module groups*, combine the simple functions implemented by single modules into complete executable subsystems.

Each module group is assigned to a *category*. One group in each installed category is said to be *active*, meaning its modules will be executed. Finally, a *configuration* is simply a specification of which group is active in each category.

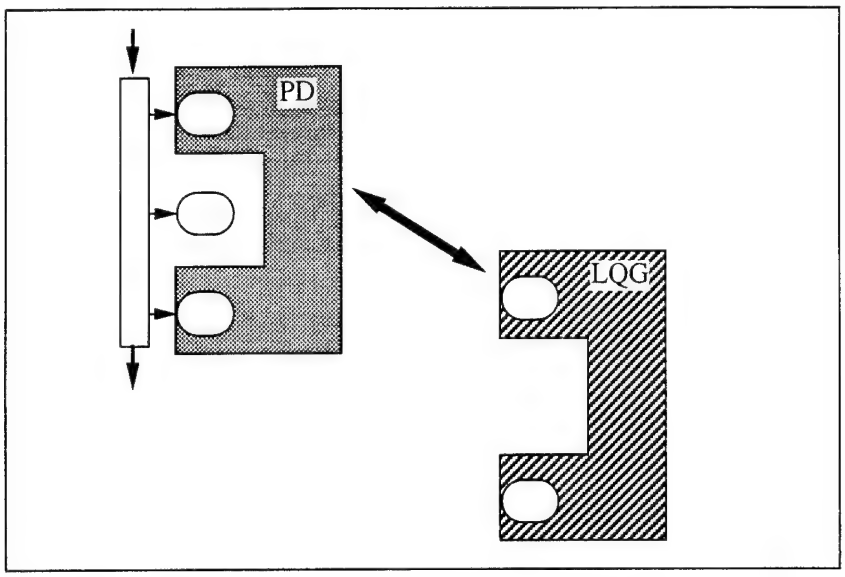


Figure 2.5: **Configuration Manager**

Configurations can be swapped in or out under program or menu control. This provides flexible run-time reconfiguration of the execution structure.

Example As a simple example, consider a system with two controllers: a proportional-plus-derivative controller named "PD", and an optimal controller known as "LQG". Suppose the PD controller requires filtered inputs, and thus consists of two sample modules: an instance of the *PDControl* component and a filter component. These two components would comprise the "PD" module group. The "LQG" controller module group may also be made up of several components. Both of these groups would be assigned to the category "controllers".

The user (or application code) can then easily switch controllers by changing the active module group in the "controller" category.

Now suppose further that the controllers require a more sophisticated sensor set. A category named "sensors" may also be defined, perhaps with module groups named "endpoint" and "joint". The highest level of the hierarchy allows the user to select an active group from each category, and name these selections as a *configuration*. Thus, the "JointPD" configuration might consist of the "joint" sensors and the "PD" controller. The "endptLQG" configuration could be the "endpoint" sensors and the "LQG" controller.

Category and Group Specification This subdivision may seem complex in these simple cases. However, it is quite powerful in more realistic systems. It has been shown to be quite natural in applications ranging from a vision-guided dual-arm robotic system able to catch moving objects [43]

to flexible-beam adaptive controllers [4].

Assigning modules to groups and groups to categories is made quite simple with the ControlShell graphical DFE editor's "configuration definition" window, shown in Figure 2.6. New categories are added with the click of a button. To create a module group, the user simply names a group, and then clicks on the modules in the data-flow diagram that should belong to that group. The blocks are color-coded to relate the selections back to the user.

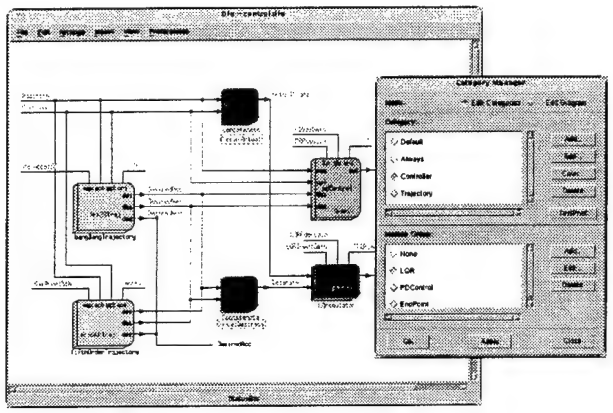


Figure 2.6: **Configuration Definition**

Configurations are easily defined within the DFE graphical interface.

2.6 Finite State Machines

A real-time system in the real world must operate in a complex, event-driven environment. With only a sequential programming language, the burden of managing and reacting to events is left to the programmer.

The Finite State Machine (FSM) module is designed to provide a simple strategic-level programming structure that also assists in managing events and concurrency in the system. The FSM module combines a non-sequential programming environment with natural event-driven process management. With this structure, the programmer is actively encouraged to divide the problem into small, independently executing processes.

To utilize the FSM module, the programmer first describes the task as a state transition graph. The graph can be directly described within ControlShell's graphical FSM editor (see Figure 2.7). Each transition—represented by an arrow in the graph—specifies a starting state, a boolean relation between stimuli that causes the transition, the CSModule to be executed when the transition occurs,

and a series of "return code-next state" pairs that determine the program flow.

The FSM model is quite general; it supports rule-based transition conditions (reducing the number of states in complex systems), true callable sub-chains of states (so libraries of state subroutines can be developed), wild-card matching (so unexpected stimuli can be processed), global matching (allowing easy error processing), and conditional succession (so state programs may easily branch). Transitions are specified as boolean relations of three types of stimuli: transient, latched, and conditional. Transient stimuli have no value, and exist only instantaneously. Latched stimuli also have no value, but persist until some transition expression matches. Condition stimuli have string values; they persist indefinitely and thus represent memory in the system. Thus, the transition condition "Object = Visible AND Acquire" might cause a system to react to an acquisition command from a high-level controller. Providing these three stimuli types allows combination of both "system status" and "event" types of asynchronous inputs into easily-understood programs.

The FSM module takes advantage of the atomic message-passing capability of modern real-time kernels to weave the incoming asynchronous events into a single event stream. Any process can call a simple routine to queue the event; the FSM code spawns a process to execute the resulting event stream. The result is an easy-to-use, yet powerful real-time programming paradigm.

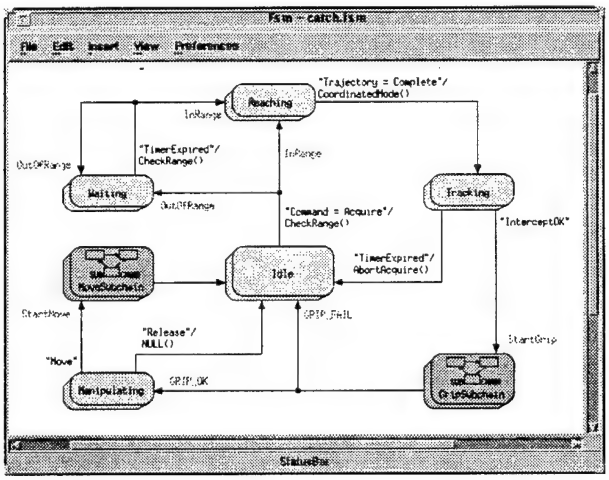


Figure 2.7: **Finite State Machine Editor**

State transition graphs allow easy visualization of multi-step operations; this example is a (simplified) program to catch a moving object with a dual-arm manipulator.

2.7 Data Control and Binding

Most data in a ControlShell application is embodied in *CSMats*. A CSMat is a named matrix of floating-point values. Each row and column of the matrix optionally contains a field name and a units specification. A complete real-time matrix mathematics utilities package is included. Components may combine multiple CSMats into structures for efficient reference and parameter passing.

The entire control hierarchies are created and bound to the correct data objects at run-time. The system is built from the graphically-generated description files produced by the DFE and FSM editors. This dynamic binding paradigm is very powerful—it combines the convenience of automatic system building with the flexibility of a dynamically changeable system. Thus, it provides the features of a full code generation without the pre-compiled inflexibility.

To support this dynamic binding, ControlShell incorporates a "linking" database facility. All instances of each data object (such as CSMats)—and each control construct (such as execution lists)—are entered into the database upon creation. The database allows "reference before creation" semantics for many object types; if a requested object is *not* in the database (i.e. it does not exist), an incomplete (e.g. zero-sized) object will be created by the database itself. This capability allows considerable flexibility at run-time; modules may, for instance, specify dependencies on data sets that do not yet exist, etc. Verification routines insure that the system is consistent before actual "live" execution begins.

2.8 Network Connectivity

ControlShell is integrated with a network connectivity package called the Network Data Delivery Service (NDDS) [162]. NDDS is a novel network-transparent data-sharing system. NDDS features the ability to handle multiple producers, consumer update guarantees, notifications or "query" updates, dynamic binding of producers and consumers, user-defined data types, and more.

The NDDS system builds on the model of information producers (sources) and consumers (sinks). Producers register a set of data instances that they will produce and then "produce" the data at their own discretion. Consumers "subscribe" to updates of any data instances they require. Producers are unaware of prospective consumers; consumers are not concerned with who is producing the data they use. Thus, the network configuration can be easily changed as required. NDDS is a symmetric system, with no "special" or "privileged" nodes or name servers. All nodes are functionally identical and maintain their own databases. The routing protocol is connectionless and "quasi-stateless[1]"; all data producer and consumer information is dynamically maintained. Thus dropped packets, node failures, reconfigurations, over-rides, etc. are all handled naturally.

This scheme is particularly effective for systems (such as distributed control systems) where infor-

[1]The databases at each node cache some state for efficiency, but all information decays over time.

mation is of a repetitive nature. NDDS is an efficient, easy-to-use distributed data-sharing system. Figure 2.8 illustrates the use of NDDS within a cooperating-arms robot system (see [33]).

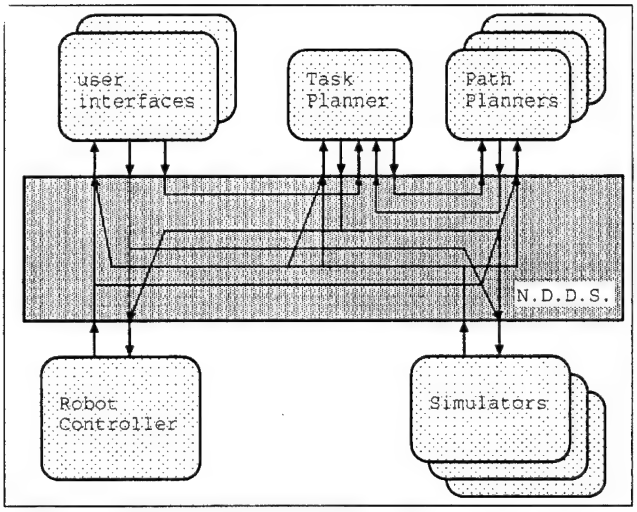


Figure 2.8: **Network Data Delivery Service**

NDDS provides a network "backplane". Each module can easily share data with any other module. The individual connections are handled transparently by the system.

2.9 Conclusions

This paper has presented a brief overview of the capabilities of the ControlShell system. ControlShell is designed—first and foremost—to be an environment that enables the development of complex real-time systems. Emphasis, therefore, has been placed on a clean and open system structure, powerful system-building tools, and inter-project code sharing and reuse.

Chapter 3

The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications

This chapter is based on the paper "The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications," by G. Pardo-Castellote and Stan Schneider, published in Proc. of IEEE Int. Conf. on Robotics and Automation, *San Diego, May 1994*

3.1 Introduction

Many control systems are naturally distributed. This is due to the fact that often they are composed of several physically distributed modules: sensor, command, control and monitoring modules. In order to achieve a common task, these modules need to share timely information. Robotic systems are a prime example of such distributed control systems.

These information sharing needs are common to many other application environments such as databases, distributed computing, parallel computing, transaction systems, etc. However, distributed control applications have some unique requirements and characteristics:

- Data transactions are often time-critical. For control purposes data must get from its source to its destination with minimum delay.
- Computations are often data-driven, that is, triggered by the arrival of new data. For instance, a collision-avoidance plan may need to be re-evaluated as soon as a new obstacle is detected.

- A significant portion of the data flow is repetitive in nature. This is true of sensor readings, motor commands etc. For this type of data, data loss is often not critical: sending data is an idempotent operation and new updates just replace old values. Considerable overhead can be avoided by using a data transfer paradigm that exploits these facts.

- There are often multiple sources of (what may be considered) the same data item. For example, a robot command might be generated by a planner module as well as a tele-operation module. Similarly there can be many data consumers. A robot and a simulator are both sinks of "command-data." The network of data producers and consumers may not be known in advance and may change dynamically.

- Data requirements are ubiquitous and unpredictable. It is often difficult to know what data will be required by other modules. For instance, force-level measurements—normally used only by a low-level controller—may be required by a sophisticated high-level task planner in the future. The architecture should support these types of data flow. Thus, vital data should be accessible throughout the system.

- Most data flow can and should be anonymous. Producers of the sensor readings can usually be unaware of who is reading them. Consumers may not care about the origin of the data they use. Hiding this information increases modularity by allowing the data sources and sinks to change transparently.

The *Network Data Delivery Service* (NDDS) has been developed to address these unique needs. NDDS provides transparent network connectivity and data ubiquity to a set of processes possibly running in different machines. NDDS allows distributed processes to share data and event information without concern for the actual physical location and architecture of their peers[1]. NDDS allows its "clients" to share data in two ways: subscriptions and one-time queries.

NDDS supports "subscriptions" as a fundamental means of communication. In the application context described, subscriptions have some fundamental advantages over other information sharing models (such as client-server or shared-memory). Subscriptions cut in half the data latency over query/response type models and it allows synchronization on the latest available information as soon as it is produced.

NDDS supports multiple information sources (producers) and users (consumers). It provides clear semantics for multiple-producer conflict resolution, provides support for and guarantees multiple update rates (as specified by the consumers). NDDS's implementation is nearly "stateless" and internally uses decaying state to ensure inherently robust communication.

NDDS is integrated into the *ControlShell* real-time programming framework [63, 47] and is being been used in several applications including the control of a two-armed robotic system [33], an underwater vehicle [57], and a self contained, two-armed space robot originally described in [56].

[1] NDDS is being used to communicate between Sun, HP and DEC workstations as well as VME-based real-time processors running the VxWorks operating system.

This paper describes the philosophies behind the NDDS system, and details an example application of a dual-arm robotic system capable of planning and executing complex actions under the control of an interactive graphical user-interface.

3.2 Relation to other research

There is a large body of literature covering information sharing in distributed computer systems [5, 8]. Systems that support some of the most popular schemes, such as shared-memory, have been developed for robotic applications [18]. The shared memory abstraction is intuitive and well understood. However, it is hard to implement efficiently in a distributed environment without special hardware [59], doesn't scale well to large systems and does not take advantage of the characteristics of Distributed Control Systems (DCS) that were listed in the introduction.

In the last few years, several data-sharing systems that take advantage of special characteristics of DCS have been developed: MBARI's Data Manager [25], CMU's TCX [11], Rice University's TelRIP [61] and Sparta's ARTSE [51] all offer network-transparent connectivity across different platforms and support *subscriptions* as means of communication where multiple consumers can get updates from a single producer. Of these, only the Data Manager provides support for multiple (consumer-specified) update rates. And only TelRIP supports multiple producers of a single data item. None of the above architectures combine the above facilities with NDDS's fully-distributed, symmetrical implementation (absence of privileged nodes) nor use a re-startable handshake-free stateless protocol.

3.3 The NDDS communication model

The NDDS system builds on the model of information producers (sources) and consumers (sinks).

Producers register a set of data instances that they will produce, unaware of prospective consumers and "produce" the data at their own discretion. Consumers "subscribe" to updates of any data instances they require without concern for who is producing them. In this sense the NDDS is a "subscription-based" model. The use of subscriptions drastically reduces the overhead required by a client-server architecture; Occasional subscription requests, at low bandwidth, replace numerous high-bandwidth client requests. Latency is also reduced, as the outgoing request message time is eliminated.

NDDS identifies data instances by name (their *NDDS name*). The scope of this name extends to all the tasks sharing data through NDDS. Two instances with the same NDDS name are viewed by NDDS as different updates of the *same* data instance and are otherwise indistinguishable to the client. If two data instances must be distinguished by any NDDS client, they must be given a different NDDS name.

Function	Action
NddsRegisterProducer	Register and specify producer parameters
NddsProduce	Add an instance produced by the producer
NddsSampleProducer	Take a snap-shot of all the instances produced by the producer. For immediate producers also send updates to all consumers.
NddsRegisterConsumer	Register and specify consumer parameters
NddsSubscribeTo	Add a subscription to a consumer. Specify a call-back routine to be called on updates
NddsReceiveUpdates	Poll the consumer for updates. Will result on call-back routines being called when applicable. Required only of polled consumers.
NddsQueryInstance	Issue a one-time query.

Table 3.1: **Functional interface to produce, consume and query data.**

Producing data involves three phases: Registering (declaring) a producer, declaring the instances the producer will produce and sampling the producer. Receiving data updates also involves three phases: Registering (declaring) a consumer, declaring the instances that the consumer subscribes to along which the action to be taken and lastly receiving the updates. The last phase is only required for polled consumers.

NDDS requires all data instances to be of a known type. NDDS has some built in types (such as strings and arrays) but most data flow consists of user-defined types. Creating a new *NDDS type* involves binding a new type-name with the functions that will allow NDDS to manipulate instances of that type.

NDDS treats producers and consumers symmetrically. Each node maintains the information required to establish communications. Producers inform prospective consumers of the data they produce. Consumers use this information to either subscribe to data or issue one-time queries. Table 3.1 lists the steps involved in becoming a producer or consumer of data.

3.3.1 Producer characteristics

A producer can be compared to a multi-channel Sample-and-Hold. It is associated with a set of object instances (similar to the signal channels) that get sampled synchronously. Sampling a producer takes a sample of the values of each data item the producer has associated with it. The data is either immediately distributed (for *immediate* producers) or saved for later distribution (*delayed* producers).

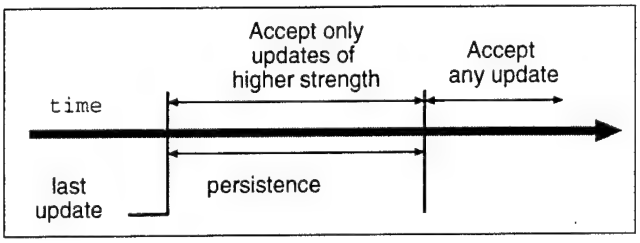


Figure 3.1: **Multiple producer conflict resolution.**

NDDS resolves the multiple-producer conflict by characterizing each producer with two properties: the producer's strength *and its* persistence. *When a data update is received for some object instance, it is accepted if either the producer's strength is greater (or equal) to that of the producer of the last update for that instance or, the time elapsed since the last update was received exceeds the persistence of the producer of the last update. In essence the strength is like a priority and the persistence is the duration for which the priority is valid.*

A producer is characterized by three parameters: *production rate*, *strength* and *persistence*. The strength and persistence parameters are used to resolve multiple-producer conflicts. Their meaning is illustrated in Figure 3.1. A producer's data is used while it is the strongest source that hasn't exceeded its persistence. Typically a producer that will generate data updates every period of length T, will set its persistence to some time T_p where $T_p > T$. Thus, while that producer is functional, it will take precedence over any producers of less strength. Should the producer stop distributing its data (willingly or due to a failure), other producers will take over after T_p elapses. This mechanism establishes an inherently robust, *quasi-stateless* communications channel between the strongest producer of an instance and all the consumers of that instance.

3.3.2 Consumer characteristics

Consumers are *notification based.* They subscribe to a set of instances (identified by their NDDS name) by providing *call-back functions* for each instance they subscribe to. When a data update for a subscription arrives, the call-back function of every consumer is called with the data as a parameter.

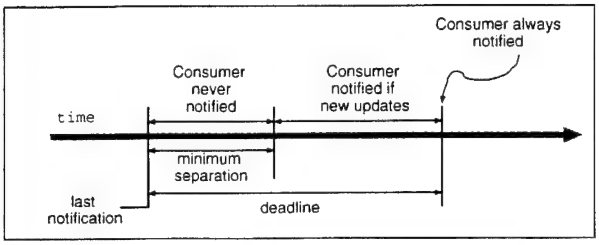


Figure 3.2: **Consumer notification rate.**

The NDDS characterizes consumers requests for periodic updates with two properties: the consumer's minimum separation *time and its* deadline. *Once the consumer is called with an update for an object instance, it is guaranteed not to be notified again of the same instance for at least the minimum separation time. The deadline is a the maximum time the consumer is willing to wait for a new update. Even if new updates haven't arrived, the call-back routine will be called when the deadline expires.*

Two consumer models are currently supported: *immediate* and *polled.* An immediate consumer will be called back as soon as the data update arrives. A polled consumer will not be called back until it itself "polls" for updates.

Consumers are characterized by two parameters, the minimum separation and the deadline (see figure 3.2). These parameters are used to regulate consumer update rates. Consumers are guaranteed updates no sooner than the minimum separation time and no later than the deadline.

Typically the minimum separation protects the consumer against producers that are too fast whereas the deadline provides a guaranteed call-back time that can be used to take appropriate action (the expiration of the deadline typically indicates lack of producers or communications failure).

3.3.3 One-time queries

A client task may issue one-time queries for specific NDDS data instances.

Queries are blocking calls. Aside from specifying the name and type of the NDDS data instance, a query contains two parameters: the *wait* and *deadline* illustrated in figure 3.3. These parameters regulate the tradeoff between returning as soon as data becomes available and waiting for "better" data. The use of these parameters make the latency of this call predictable, allowing its use from real-time application code. Typically the wait is set to be long enough to account for communication delays from all data producers to the consumer. The deadline provides a guaranteed call-back time in case no responses arrive. Setting a wait time to zero causes the first response to be accepted.

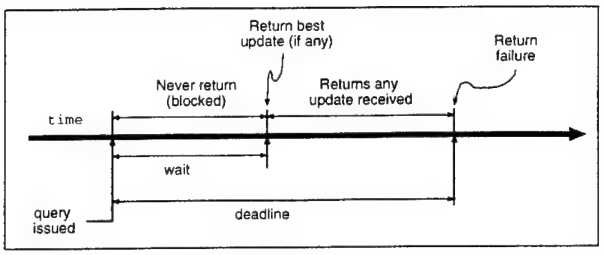


Figure 3.3: **One-Time Query Parameters.**

A one-time query specifies two parameters: the wait *time and the* deadline. *A query will block for at least wait time during which the data arriving from the producer with higher strength is saved. At the end of the wait time the query returns if any data has been received. Otherwise, query remains blocked until either an update comes or the deadline expires (whichever comes first).*

3.3.4 Reliable updates

By default NDDS provides unreliable, unacknowledged data updates from producers to consumers. While this gives the most throughput with minimum overhead, it may result in occasional loss of updates or out-of-order arrival. For sensory-type data, having the latest data as soon as it becomes available ,is usually more important than occasionally missing an update. Other kinds of information, such as commands, often present in distributed control systems, would be better served with a reliable protocol.

NDDS supports reliable updates. A producer may specify any one of its productions to be delivered reliably. Reliable updates get grouped together in special packets that are individually acknowledged. The producer is notified if the update isn't acknowledged by a specified deadline and/or if another reliable production is attempted before the last one was acknowledged. This guarantees in-order update delivery at the expense of reduced update bandwidth. A window size larger than one may be specified to compensate for longer communication delays so that multiple reliable updates can be sent before any acknowledgement is received.

3.4 Implementation

NDDS is symmetrically distributed, that is, there are no "special" or "privileged" nodes nor name servers. All NDDS nodes are functionally identical and each node maintains its own copy of the NDDS database and contains the helper processes necessary to implement the communication model described above.

NDDS uses UDP/IP [37, 9] as a means of communication. To allow communications between computers with different data formats the External Data Representation (XDR) [26] is used.

3.4.1 Architectural overview

An NDDS *node* is composed of one or more NDDS client processes (each with its respective NDDS Server Daemon) a copy of the NDDS database and three daemon (helper) processes that maintain the database and implement the NDDS communication model described above. See figure 3.4.

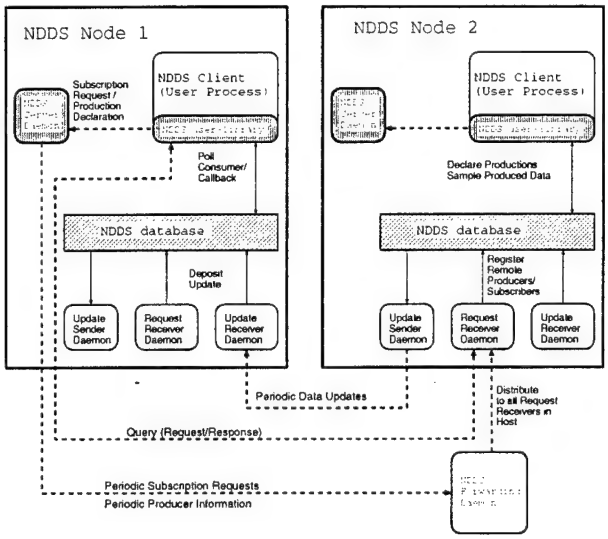


Figure 3.4: **Communication between NDDS nodes.**

A NDDS node *is composed of one or more NDDS* clients *and three helper daemons that share a local copy of the NDDS database. Each NDDS* client *has a private NDDS Server Daemon that informs other NDDS* nodes *of the productions and subscriptions of the client. The three helper daemons are responsible for maintaining the NDDS database, sending updates to remote subscribers, receiving updates and servicing queries. There is one NDDS Forwarding Daemon per Network host.*

The user task becomes an NDDS client by linking to the NDDS library. Each NDDS client process spawns a private NDDS Server Daemon process that will assist in establishing the subscriptions and informing the peer nodes of the user productions. There is at most one NDDS node per address space so in operating systems that support shared memory threads (for example VxWorks), several NDDS client processes may belong to the same node (sharing the same copy of the NDDS database

and helper daemons[2]).

The following is a functional description of the different daemons:

- **NDDS Forwarding Daemon** (NFD). There is one NFD per network host. All the Request Receiver Daemons running on the host register with the NFD. Production notifications and subscription requests received by the NFD daemon are immediately forwarded to all the Request Receiver Daemon(s) running on the host.
- **NDDS Server Daemon** (NSD). Each NDDS client (user-task) spawns its private NSD. The NSD is responsible for periodically informing the all the other NDDS nodes of both the subscription requests and the productions of the NDDS client.
- **Request Receiver Daemon** (RRD). There is one RRD per NDDS node. The RRD is responsible for maintaining the remote subscriptions and productions in the NDDS database. Stale productions and subscription requests are aged and eventually dropped by the RRD. This daemon is also responsible for replying to one-time *queries* from other NDDS nodes.
- **Update Sender Daemon** (USD). There is one USD per NDDS node. The USD is responsible for sending the updates of locally produced data items to the subscribers in other NDDS nodes. This daemon also ensures that the the timing parameters requested by the consumer are met.
- **Update Receiver Daemon** (URD). There is one URD per NDDS node. The URD is responsible for receiving updates for the local subscriptions of the nodes. The URD solves multiple-producer conflicts and, in the case of *immediate* consumers, executes the callback routine(s) installed for that data item. The URD also ensures that the timing parameters requested by the each consumer in the node are met.

3.4.2 Data management overview

The NDDS database is replicated and maintained on each NDDS *node* by three helper daemons (the Request Receiver Daemon, Update Sender Daemon and Update Receiver Daemon). The database stores and cross references producers and the data they produce, consumers and the data they consume, remote productions, subscriptions requested by both the NDDS clients in both the local and remote NDDS nodes, etc.

Consistency between databases across different NDDS nodes is not necessary and requires no special effort. Temporary inconsistencies between databases may result on subscription requests (or queries) not reaching all the producers of a given data item and, as a consequence, different nodes may get data from different producers. A similar situation may result from the data loss due to

[2]In operating systems that don't support shared memory threads, such as Unix, the helper daemons are not independent tasks but rather are installed as signal handlers.

communication failu . At worst this will be a transient situation that arises only if there are multiple producers o the same data.

All information about remote NDDS nodes is aged and is eventually erased unless it is refreshed. The NDDS Server Da ion associated with each NDDS *client* is responsible for the periodic refreshment of the information that concerns that NDDS *client* . This mechanism is inherently robust to remote node failures, communication dropouts and network partitioning. Furthermore, it requires no special recovery mechanisms.

3.5 Applications

This section describes how a distributed robotic application exploits NDDS unique facilities to build a modular expandable system that integrates planning into a two-armed robotic work-cell [33]. The system is composed of four major components: user interface, planner, the dual-arm robot control-and-sensor system, and on-line simulator. The graphical user interface provides high-level user direction. The motion planner generates complete on-line plans to carry out these directives, specifying both single and dual-armed motion and manipulation. Combined with the robot control and real-time vision, the system is capable of performing object acquisition from a moving conveyor belt as well as reacting to environmental changes on-line.

The use of NDDS as the underlying communication mechanism provides unique benefits to this application without requiring any special programming.

- The different modules can be distributed across different computer systems (with different processor architectures and operating systems)[3].
- Several copies of the *same* module can be run concurrently. For example, the graphical user interface module can be run on several workstations. This allows multiple users to monitor the system and permits *simultaneous* interaction with the robot system.
- The graphical simulator module can *masquerade* as the real robot and allow independent testing of the remaining modules in the system. Any time the real robot goes on-line, its productions override those of the simulator[4] and all the remaining modules are now connected to the real robot.
- Should the planner be unable to generate adequate plans for specific situations due to limitations or malfunctions, a teleoperation module (under development) can override planner commands and take control of the robot.

[3]This experiment has modules running on DEC Workstations, Sun Workstations and several VME-based real-time processors.

[4]Thanks to NDDS's multiple-producer conflict resolution mechanism.

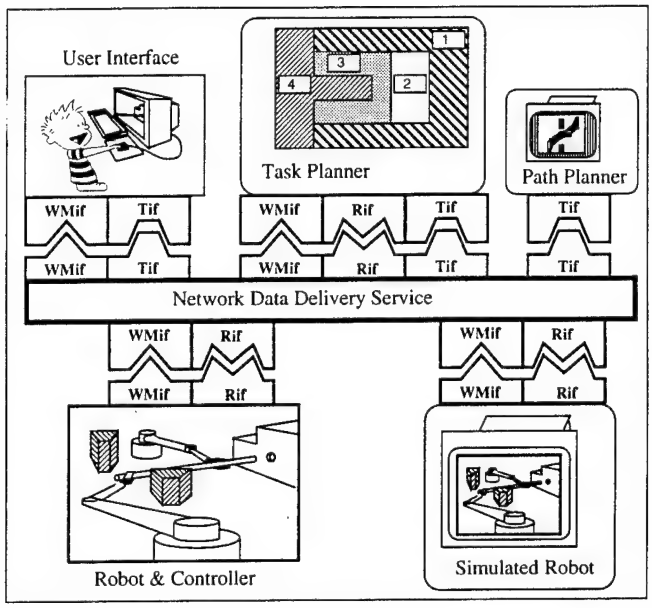


Figure 3.5: **Application Example: Task-Level control of a Two-Armed robot.**

This example shows the main application modules. Each module communicates using one or more of the three interfaces: The World Model interface (WMif), the Robot Interface (Rif) and the Task Interface (Tif). These modules are physically distributed. All the interfaces are built using the Network Data Delivery Service (NDDS). NDDS plays the role of a bus providing the necessary module interconnections.

- The modules can be started in any order. Future modules (such as the teleoperation-module mentioned above) can be *dynamically* added to the system even if it is already in operation or deployed[5].

3.6 Conclusions

This paper has presented NDDS, a unique data-sharing scheme that allows programs distributed on a computer network to share data and event information unaware of the location of their peers. These facilities provide fundamental new capabilities to distributed control systems that use NDDS as the underlying communication mechanism. This paper has also discussed an application that uses NDDS to communicate between different modules that integrate planning into a two-armed robotic work-cell. Several other applications are cited in the paper.

[5]This facility may prove crucial for other current applications such as the undersea vehicles.

Chapter 4

System Design and Interfaces for Intelligent Manufacturing Workcell

This chapter is based on the paper "The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications," by G. Pardo-Castellote, Stan Schneider, and R.H. Cannon Jr., published in Proc. of IEEE Int. Conf. on Robotics and Automation, *Nagoya, Japan, May 1995*

4.1 Introduction

Integration of a robotic workcell has traditionally been a very time-consuming process. The result is higher costs and longer lead times. As a consequence, the number of applications for which robotic automation is economical remains small.

Some of the reasons for the complexity of the integration are:

- Controlling the robotic workcell and connecting it to the rest of the manufacturing facility requires a large software development effort. The software is usually structured as many individual modules with a high degree of interconnection: interface and monitoring stations, scheduling, control, sequencers, planning subsystems etc.
- A lot of custom mechanical "glue" is required to ensure predictable, repeatable behavior of the workcell: feeders, fixtures, specialized tooling and fixed automation.
- There is a large lead time before initial system operation, and each design iteration is slow since it requires repositioning the fixtures, teaching new robot trajectories, etc. As a result, a lot of time needs to be spent on the early design phase and there is little room for iteration.

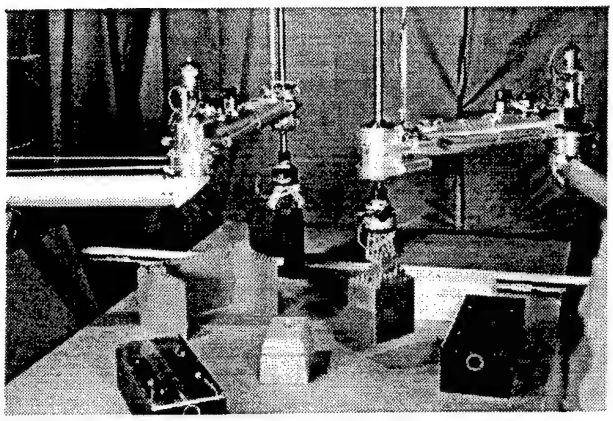


Figure 4.1: **The Dual-Arm Workcell**

This workcell is capable of performing vision-guided moving-object acquisitions and assemblies under the control of an on-line planner.

Mechanical glue and custom integration time can be reduced by making the cell smarter and sensor-rich so the workcell uses less structured feeding techniques (e.g. conveyors) and can plan and schedule trajectories autonomously (avoid teaching, and scheduling phases), however this exacerbates the software development and integration part.

This paper describes a novel system design and integration approach taken at the "Smart Robotic Workcell Project" at Stanford. It analyzes and addresses these problems and presents experimental verification of the system design in operation.

Our system relies in the use of visual tracking and on-line planning to avoid the need for fixturing or a priory scheduling. The on-line planning and control issues have been described in [22, 44]. The real-time vision system is described in [43]. This paper focuses on several new technologies developed to address the integration problem;

1. A new "interfaces-first" system-design technique (similar to the approach taken in large software projects) is emphasized over the "components-first" design technique commonly used for robotic systems.

2. Modular, parametric interfaces. The interfaces themselves are designed to be composed of multiple primitive modules which can be customized and connected in a variety of ways to create new "custom" interfaces for any subsystem modules.

3. Anonymous, stateless interfaces. The interfaces do not specify the subsystems involved, and messages are self-contained, increasing reliability and enabling replication and arbitrary connectivity.

Section 4.2 gives a brief description of the hardware and the experimental context that motivated the research, section 4.3.1 describes the new approach to system design based on the idea of starting with the interfaces (and making these modular themselves), section 4.3.3 describes the primitive interfaces used in the "Smart Robotic Workcell" project, and section 4.3.4 describes the resulting system architecture and several of the configurations that can be achieved using the primitive interfaces. The paper concludes with some experimental results on the the system's operation.

4.2 Overview of the Experiment

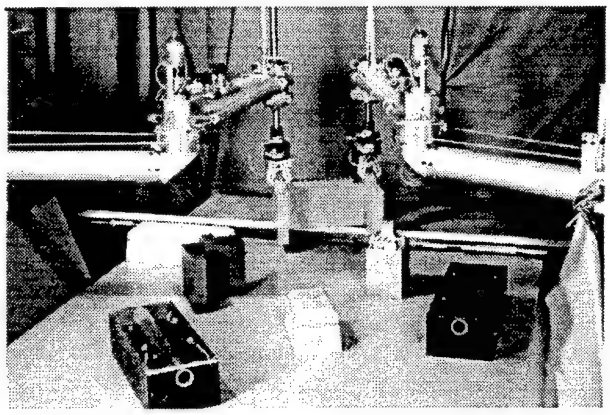


Figure 4.2: **Dual-Arm Motion**

The system can use both arms simultaneously, either independently or to perform cooperative motions, as pictured here.

The "Smart Robotic Workcell" illustrated in figure 4.2 is composed of two 4-DOF arms in a SCARA configuration, a conveyor belt, an overhead vision system, and several objects that can be grasped and manipulated by the arms. Both the arms and the objects are tagged with infra-red LED's and are tracked using an overhead vision system that uses the unique 3 LED signature of each object to identify and track both the position, orientation and velocities at 60 Hz.

The experimental context of this project is the development of an architecture that allows semi-autonomous operation of the workcell. In the experiments performed, the human specifies a high-level assembly task using a graphical interface. The assembly is specified as the relative position and orientation of the different parts with respect to the "assembly reference frame". The user also specifies the location of the assembly within the workcell.

The objects required for the assembly may either be present or brought in on the conveyor. Some of the objects can be grasped with a single arm while others (due to their weight or shape) require

dual-arm manipulation. Additional objects that cannot be moved by the robot constitute obstacles that the robot must avoid.

Once the goal is specified, the user need not interact with the system other than for monitoring purposes. The system will use on-line planning to generate and execute capture trajectories as objects appear on the conveyor, and deliver the objects to their desired goal locations. The delivery of objects may require multiple steps such as regrasps or hand-over of the objects as well as trajectories that avoid obstacles that may move. The system can use both arms cooperatively to manipulate an object with both arms (figure 4.2), or independently to increase the throughput of the workcell by capturing two objects simultaneously (figure 4.1), or capturing an object with one arm while the other arm is moving.

Experimental robotic systems with similar objectives have been described by several authors [13, 1], while other authors have focused on developing generic system architectures and their interfaces [23, 12, 29, 62, 27, 34, 38, 36]. Here we will compare ours with a few representative systems from the view point of the architecture and interfaces.

The KAMRO workcell [1, 15] combines an on-line planner (FATE) and a real-time control subsystem. The input to FATE is a petri-net describing the assembly task. FATE interprets the net and sends elementary commands (transfer, fine-motion, grasp) to the controller, and monitors its progress. Our approach is similar except the commands generated by the planner are *strategic commands* which are further decomposed into elementary operations by a finite-state machine facility *within the control subsystem.* Monitoring and error-recovery occurs in both in the control and planning subsystems. This extra layer allows our system to deal with moving objects and obstacles.

RCS and NASREM [38, 3] are strictly hierarchical approaches to system design. For telerobotic applications, UTAP [23] provides a more detailed specification of individual subsystems and their interfaces. UTAP resembles subsystems-first design in that all modules accept similar messages, however, in UTAP interfaces have state (messages are processed in the context of previous messages), message exchange isn't anonymous (sender specifies receiver), there is limited connectivity (strictly hierarchical messages), and expandability (the type and number of modules is fixed). UTAP's messages are richer because they include language constructs (macros, message groups etc.).

Object-Oriented programming (OOP) facilities are well suited for interfaces-first designs and the construction of modular interfaces. Primitive interfaces can be encapsulated in individual abstract classes, and multiple inheritance used to build dedicated interfaces from the primitive elements. However, most uses of OOP in robotic applications use objects to model the individual subsystems, not the information that needs to be exchanged (subsystems-first approach). For instance, RIPE [27] defines an object hierarchy to describe the different physical elements in the system. In RIPE, class definitions become in effect interface specifications. For instance, the robot class specifies methods to move to a point, move along a certain axis, move along a path, close and open grippers etc. RIPE does not define interfaces for sensor or world state information, nor task specification.

4.3 Interface Design

The importance of interface design cannot be overstated:

> The greatest leverage in system architecting is at the interfaces. *Eberhardt Rechtin* [42]
> The greatest dangers are also at the interfaces *Arthur Raymond*[USC, 1989] [42], pp. 89

4.3.1 Alternative approaches to system design

Modular system design is a well-established methodology. This approach breaks a large system into smaller, well-defined modules with specified functionality, which then can be developed and tested independently from the others.

While modularity facilitates the development of the individual parts, the need to develop the interfaces [glue] between parts makes the complexity of the overall system much greater than that of the sum of its parts. Hopefully by careful selection of the functional boundaries, the resulting subsystems are somewhat independent and the total complexity can approach the ideal limit of the sum of its parts.

Subsystems-First Traditional modular system design (*subsystems first*) follows the cycle illustrated on the left side of Figure 4.3. Following the analysis of the complete system, functional units are identified and the system is broken into subsystems across these functional boundaries. These subsystems are further defined by their interfaces to the outside world, which specify their observable behavior. Once the subsystems have been developed and individually tested, custom interfaces (glue logic) are developed to interconnect the different subsystems. This results in a complete system that can now be tested against the design specifications. This process repeats itself on each design iteration.

This *subsystems-first* technique requires that the scope and functionality of the overall system is known in advance. It emphasizes subsystems rather than their interfaces. It is therefore most appropriate for "sparsely-interconnected" one-of-a-kind systems that, once designed, have little need to expand and add new subsystems. For example, a monolithic electronic board (say a graphics board) is designed as a unit from the onset and the different functional units (CPU, video memory, graphic accelerator hardware, etc.) are designed with their own interfaces and require custom glue-logic to be connected to each other within the board. This approach is most appropriate for systems where subsystems have low degree of interconnectivity.

Interfaces-First There are other systems, however, where the full scope isn't known in advance and/or they require a much higher degree of interconnection. These are better designed using the *interfaces-first* methodology, where the types of information flow between the subsystems (whatever they turn out to be) are identified from the onset and interfaces for the information (*not* the

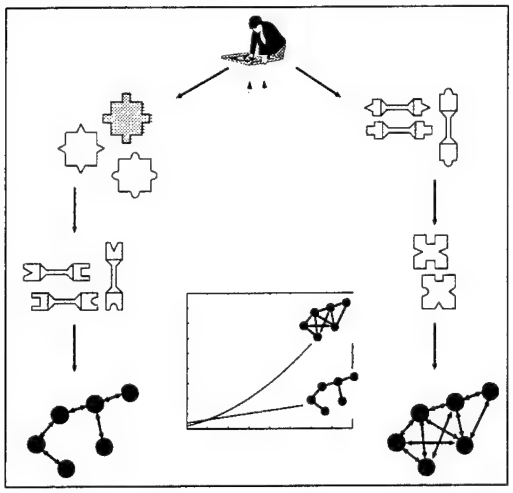


Figure 4.3: **Contrasted Design Techniques**

Subsystems-first design, on the left, concentrates on developing functional units. Information flow between the units requires custom interfaces. Interfaces-first design on the right starts with then information flow. Functional units are then tailored to the interfaces developed. Interfaces-first design is more appropriate when subsystems are highly interconnected.

subsystems) are designed first. Later, subsystems are designed that use these interfaces to communicate and interact. This design is more typical of large software projects or computer systems. For example the busses in different computer families (e.g. PC bus, VME bus etc.) are designed first, and the different boards that interact through these busses are designed later to conform to the interface specification. In fact, new boards are designed at a later stage with functionality that wasn't anticipated. This design cycle, shown down the right side of Figure 4.3, is most appropriate for "heavily interconnected" systems and those that are "open-ended" in the sense that new pieces will have to be incorporated at a later stage.

Traditionally, robotic systems have been designed with a subsystems-first technique. That has made them hard to develop and extend due to their limited interconnectivity and the considerable amounts of custom interfacing required.

4.3.2 Design of robotic interfaces

Design of the subsystem interfaces is not a trivial task for robotic systems because the information/command flow is much more structured (i.e. higher level) and varied than that modeled with a bus-type interface. For example, all the devices connected to a SCSI bus are expected to be able to operate at (almost) the same speed and accept the same type of commands. This isn't a good

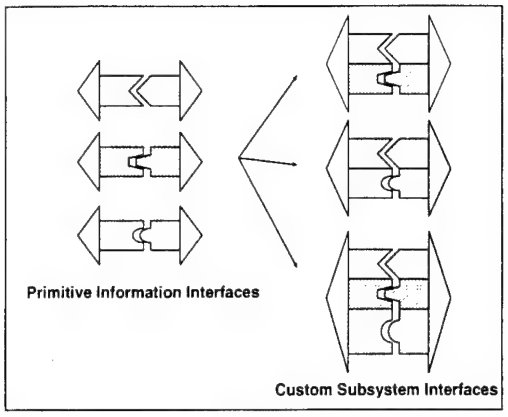


Figure 4.4: **Modular Interface Design**

First, the fundamental types of information flow are identified, and interfaces designed for each (on left). Subsystem interfaces are then built from combinations of these primitive information interfaces.

model when the "devices" are things like graphical interfaces, teleoperator-input devices, planning subsystems, and sensor subsystems. Each of these subsystems requires different fundamental types of information and is able to process it at significantly different speeds.

We propose to address this problem with a "modular" approach to designing the interfaces themselves. In particular, we first identify the fundamental properties of the information flow. We then divide the information into types based on the characteristics of the flow itself (bandwidth, persistence, idempotency). Next, we design primitive interfaces for each type of information flow. These primitive interfaces can then be combined to create custom "compound" subsystem interfaces. Figure 4.4 shows this process.

4.3.3 Primitive interfaces for robotic workcell

In the context of a robotics workcell, we must examine world-model (state) information flow, command flow, control signals, etc., identify parameters that characterize the information, and build primitive interfaces around each information type. This section examines the design of the system architecture for the smart workcell project using this interfaces-first technique.

Three types of information flow can be distinguished in this application: world state, system commands, and high-level task descriptions. Table 4.1 summarizes the three primitive interfaces and selectable parameters for these interfaces, and gives an overview of the specific information that is encapsulated within each interface.

World model state information includes data such as positions and velocities of the different objects

interface	*nature of the information*	*selectable parameters*
world state interface	periodic, idempotent, unreliable, last-is-best, persistent until the next update	update rate, specific information desired
command interface	asynchronous, reliable, in-order delivery (exactly once), persistent until completed	priority (to resolve conflicting requests)
task interface	asynchronous, persistent until cancelled, reliable but not necessarily in-order	priority

Object information from World Model	
location	object location in the global reference frame.
grasps	possible grasp locations within the object
properties	mass, inertia, limits on acceleration etc.
shape	shape (collision avoidance, graphics)

Robot information from World Model	
location	robot-base location within workcell.
joint val.	value of the joint coordinates (pos, vel, acc)
limits	kinematic (joint) and torque limits
kinematics	Denavit-Hartenberg parameters
state	moving, grasping an object etc.

Robot commands through the command interface	
move object	Move an object that is being grasped. This command will provide a via-point collision-free path for the object. The robot is controlled using object impedance control [44].
move and release	Same as above with the addition of a specified release location (must be close to the end of the trajectory)
move arms	Move one or both arms. The arm(s) to move can't be holding an object. Specifies a via-point collision-free path both in operational and joint space so that the robot controller is free to use different control schemes (and kinematic ambiguities can be resolved).
move and grasp	Same as above with the addition of the specification of the object(s) to be grasped and the corresponding arm(s), and grasp(s) location(s) for each arm involved (any combination of arm/object, including both arms on the same object is allowed)

Task commands through the task interface	
place	Place a set of objects at their specified

(on and off the conveyor), arm locations, joint angles, and force signals. By its very nature this information is either *static* or *periodic* and needs to be updated continuously, because it corresponds to physical quantities that change over time. In addition, the information is *persistent* until it is invalidated by a new update of the same information.

World model state information also has *idempotent* semantics—getting two identical updates of the same information (say the position of an object) causes no side-effect and is logically equivalent to a single update. Moreover, the natural semantics are *last-is-best*, i.e. we are always interested in the most recent value, even at the expense of missing intermediate values. The information content and frequency of each specific instance of the interface must be *customizable* since different subsystems require different subsets of the overall world-model information at widely different rates. For instance, a slow graphical interface animating the robot position may need new joint angles at only 10 Hz; a control or estimator subsystem will need updates at a much higher rate.

System command information is quite different. For one, commands are *asynchronous* and "hold their value" only until they are completed (even if no new command is received). Commands are *not* idempotent and need to be delivered exactly once reliably and in-order. For instance, if we are sending a trajectory to the robot, sending the trajectory twice will cause the robot to execute two motions. Similarly, missing an intermediate command that, say, closed a gripper before lifting an object is not acceptable.

High-level task commands fall somewhat between these two extremes. They are also *asynchronous.* However, tasks are in a sense self-contained (i.e. specify a goal or desired system state) as opposed to a process. As a result they may be *persistent.* For instance, a task may specify to repeatedly pick objects from a conveyor, immerse them in a solution, and then place them on a second conveyor. This task doesn't end until explicitly cancelled. Multiple concurrent tasks may be active at any one time and compete for the system's resources. Tasks may also be built as logical combinations of smaller subtasks (e.g. put object O1 at location L1 and object O2 at location L2).

These three primitive interfaces constitute the information building blocks (modules) from which custom interfaces can be built to connect the different subsystems. These interfaces are *anonymous* making no assumptions about the number or the specific subsystems that will be connected through them. Instead, they provide a mechanism for subsystems (whatever they end up being) to share and communicate information. Since they encapsulate the information required to interact and command the robotic workcell, they provide sufficient coverage for subsystem interaction.

On occasion a new subsystem may be developed that requires a different type of information, or a new sensor will be added that provides some other type of information. This will require a new information interface primitive. However, they should not affect the existing interfaces, except for the few new tasks and commands that utilize the new information.

4.3.4 System architecture

The basic system architecture is illustrated in figure 4.5. The overall system has been broken

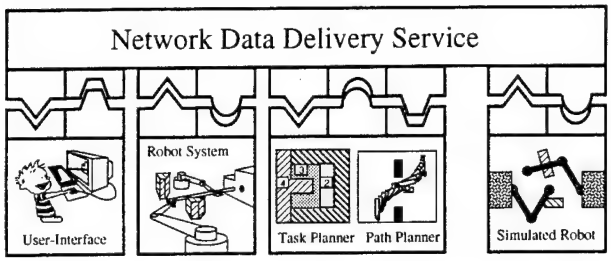


Figure 4.5: **System Architecture**

Four subsystems collaborate to operate the workcell. Each subsystem uses a combination of the primitive information interfaces to connect to the information flow. The interfaces are built using a "software bus" called the Network Data Delivery Service (NDDS).

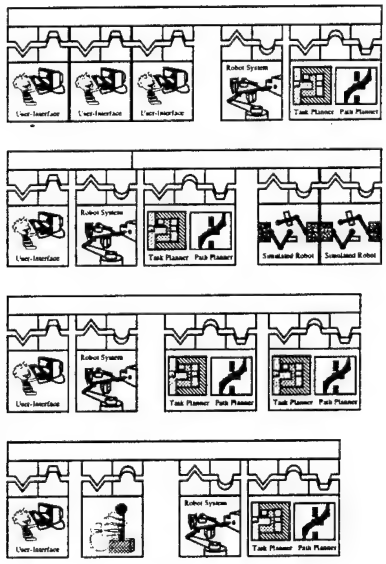


Figure 4.6: **Software Configurations**

The anonymous interfaces allows replicated modules (top three lines) in the system. The modular information interfaces allow new subsystems to be added to the system (last line).

into four subsystems: Human-interface, robot control system, task and path planners, and the robot simulator. The information primitive interfaces are represented by connection icons: state information (triangle icon), task requests (trapezoid icon), and commands (semicircle icon). The graphical human interface receives state information and provides a graphical representation of the scene to the user. During operation, the high-level task is specified by the human interface and sent to the planners. The task planner/path planner receives continuous updates from the robot and task requests from the user, and produces robot commands to implement the requested tasks. The robot controller executes these commands and processes the sensor signals to create and maintain a world model. The simulator can masquerade as the robot control system during development or, by running it concurrently with the robot, can be used to compare the accuracy of the workcell models (kinematics, dynamics and state transitions) with the behavior of the actual system.

The three primitive interfaces are used to communicate between the different subsystems. For instance, the human-interface uses the world-model interface to obtain the kinematics and position of the arms and workspace objects to display them to the user. Tasks are sent to the planners through the task-interface. The planners also use the world-model interface but at a lower bandwidth.

The different subsystems are configured as if connected to a "software bus" where the three primitive interfaces provide the means for the bus-access protocol. This allows functionally identical subsystems to be replicated and new subsystems to be added to create different configurations. The actual implementation of the software bus over the network uses a subscription communications system called the Network Data Delivery System (NDDS) [31, 41].

Information interfaces together with the software bus allow combinations of modules to be easily used. Figure 4.6 illustrates some of these configurations. In the first three cases, we have replicated modules mentioned earlier. The top line indicates several users collaborating to control the robot, or monitor its activities. The second line the use of multiple simulators to simulate different aspects of the system, such as for command previewing. The third line shows the use of.combined multiple planning strategies, each of which may be more appropriate for certain tasks. Finally, the last line indicates that modular interfaces allow us to build new interfaces such as one for a teleoperation subsystem.

4.4 System Operation

The workcell can perform autonomous multi-step operations such as the one represented in figure 4.7. Several points are worth noting: First the planner has continuous access to the state of the workcell though the world-model interface. This state is used both to maintain its internal data structures and performs precomputations, and to monitor completion/failure of issued commands (the stateless nature of the command interfaces precludes any acknowledgments; all feedback to the planner must occur via explicit observation of system behavior). Second most of commands issued by the planner are strategic, requiring the workcell to implement its own event-driven sequencing and monitoring, for instance, the `move and grasp` command is decomposed by

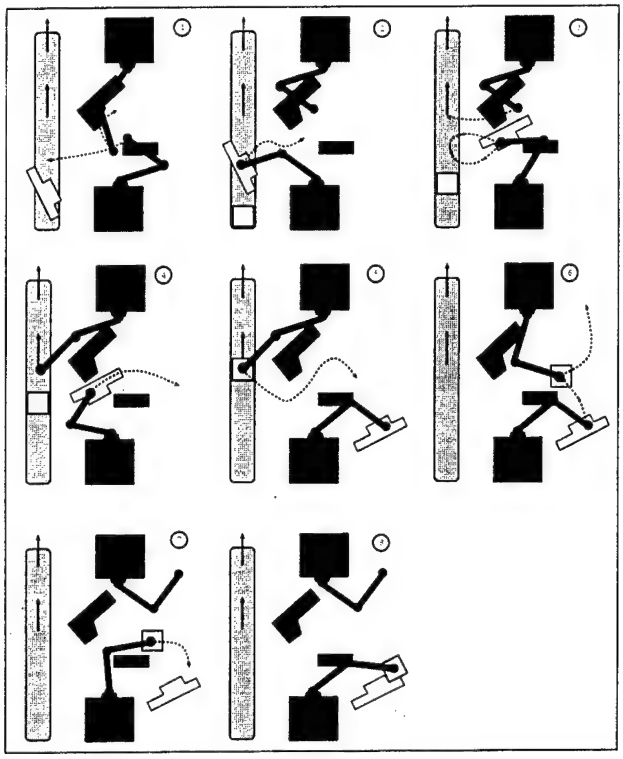


Figure 4.7: **Typical system operation**

The user has specified the simple assembly of two objects depicted in the bottom-right (8) by dragging the iconic representations of the objects on a graphical interface (not shown). After this the remaining actions are autonomous (left to right, and top to bottom): (1) The planner constantly monitors the workcell using the world-state interface and as soon as a needed object appears on the conveyor, a capture trajectory is planned and sent to the robot [`move and grasp` *command]. This command specifies the top arm to be moved out of the way, and the bottom arm to grasp the object. (2) Once the object is grasped, the planner (which detects that through the world-state interface) plans a delivery trajectory and sends it [*`move and release` *command]. The object is placed at an intermediate location where the arm can change handedness. (3) In the meantime, a new object has appeared on the conveyor, so the planner issues a* `move and grasp` *command for both arms (one for the conveyor object, the other for the just-released object). (4) While one arm picks the second object from the conveyor, the other delivers the first object to its destination [*`move and release` *command]. (5) Once the second object is grasped, since the final destination is only reachable by the other arm, the grasping arm is commanded to place it at a location reachable by both arms [*`move and release`*]. (6) Next a* `move and grasp` *command moves the first arm away while the second picks the object, and finally, (7) a* `move and release` *command delivers the object to its final destination (8).*

the strategic-workcell controller into the following command sequence (figure 4.8): `approach` (move along planned trajectory), `intercept` (move under a locally generated intercept trajectory that is constantly refined), `track, descend, grasp` (regulate directly from the object position and velocity while descending and closing the grippers), and `lift`. These actions require high-bandwidth world state updates and immediate reactive feedback, and require several control mode switches, see [46] for details. Figure 4.8, also shows data from the experimental system in operation.

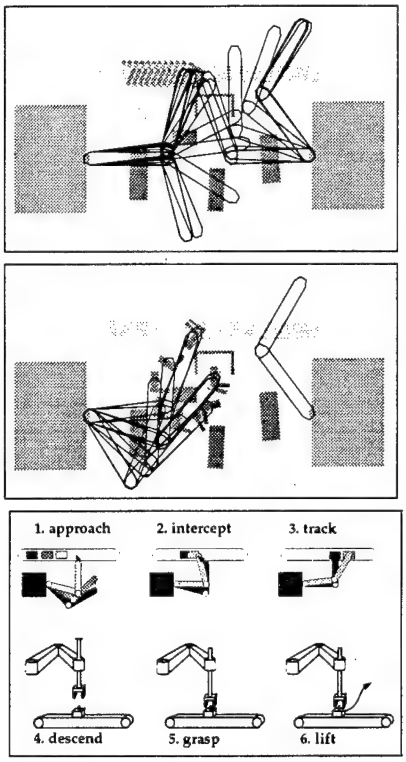


Figure 4.8: **Animation of system operation**

The top two figures are animations at 0.8 sec intervals of data collected during the capture (`move-and-grasp`*) and delivery (*`move-and-release`*) of an object. The bottom figure illustrates the stages orchestrated by the strategic controller during the* `move-and-grasp` *command.*

4.5 Summary and Conclusions

We have presented a novel approach to designing complex robotic systems called *interfaces-first* design. Interfaces-first design first identifies the fundamental types of information flow in the

system, and then encapsulates that flow into primitive anonymous interfaces. Subsystem interfaces are then built from combinations of these primitive information interfaces. This approach results in expandable systems with facile interconnectivity.

This design technique has been implemented with a network "software bus", and demonstrated on a dual-arm robot workcell. The workcell is capable of accepting high-level user task requests, planning motions to carry them out, acquiring objects from a moving conveyor, and performing simple assemblies.

Chapter 5

On-Line Manipulation Planning for Two Robot Arms in a Dynamic Environment

This chapter is based on the paper "On-Line Motion Planning for Two Robot Arms in a Dynamic Environment," by T.Y. Li and J.C. Latombe, which will appear in The Int. J. of Robotics Research.

5.1 Introduction

Off-line robot motion planning is a one-shot computation prior to executing any motion. It requires all pertinent data to be available in advance. In contrast, on-line planning is an ongoing activity that receives a continuous flow of information about events occurring in the robot environment. While planned motions are being executed, new plans are generated in response to incoming events.

Off-line planning is virtually useless in dynamic environments that involve events whose occurrences in time and space are not precisely known ahead of time. On the other hand, while on-line planning can potentially deal with such environments, it raises difficult temporal issues which have not been thoroughly addressed by previous research. Indeed, timing is highly critical since motions must be *both planned and executed* while their goals are still relevant. Opportunities to achieve a goal may exist only during short periods of time. If the on-line planner is too slow or does not focus on the right subproblem at the right time, it will fail to achieve goals that could have been attained otherwise.

In on-line planning, failing to achieve some goals is acceptable. After all, if new events can occur at arbitrary rate, there might be no way for the robot system to react timely, even if it had

unlimited computational power. The efficiency (or competitiveness) of an on-line planner should be measured relative to an instantaneous oracle having full knowledge of future events and making the best decision at every time [159]. The greater the efficiency, the better the planner.

In this chapter we investigate on-line motion planning in the context of a specific, but practical part-feeding scenario where two robot arms must grab parts as they arrive on a conveyor belt and transfer them to given goals without collision. This scenario is typical of workcells in which robots load machines, assemble products, or package/palettize parts. Today, imprecise events are eliminated by costly engineering and/or handled by enforcing time-consuming coordination rules. On-line motion planning has the potential to significantly reduce the development time and implementation cost of these cells, while increasing their throughputs. Moreover, since the timing of the operations no longer requires off-line prior analysis, cells can also be more flexible; for instance, they may be dynamically assigned new tasks without interrupting current operations.

Our approach to on-line planning is to break the overall planning problem into a series of subproblems and orchestrate very fast primitives solving these subproblems according to the incoming flow of information. We have implemented a planner embedding this approach and have experimented with it in a simulated environment to evaluate its efficiency against quasi-optimal oracles. The results show that it is quite competitive. We have also connected the planner to a real dual-arm robot system and successfully run experiments with this integrated system.

In our scenario, the transfer of a part to its goal may require "hand-over" operations between the two arms, i.e.: an arm may have to ungrasp the part at an intermediate location (e.g., because the goal is not reachable by the arm), where it will later be regrasped by the other arm. Therefore, the planner must not only compute arm motions. It must also include grasp/ungrasp/regrasp operations between these motions. For that reason we call it a *manipulation planner.*

Section 2.2 relates our work to previous research, reviews motion planning concepts used in the rest of this chapter and stresses the contribution of our work. Section 2.3 describes the part-feeding scenario that we use to investigate on-line manipulation planning. Section ?? gives an overview of our planner and Section ?? describes in detail the techniques it uses. Section ?? presents several extensions. Section ?? describes the implementation. Section ?? provides measures of its efficiency in a simulated robot environment. Section ?? reports on the connection of the planner to a real robot system.

5.2 Relation to Previous Work

Motion planning has attracted a great deal of interest over the last 15 years. Most of the research, however, has focused on off-line planning in static environments. A plan is then computed as a geometric *path.* An important concept produced by this research is the *configuration space*, or *C-space*, of a robot [144]. Various path planning algorithms based on this concept have been proposed [130]. A number of very fast planners have been implemented for robots with few degrees of freedom (usually, 3) [68, 69, 140]. A typical technique consists of exploring a uniform grid in C-

space, using a best-first search algorithm guided by a goal-oriented potential field [68]. Reasonably efficient planners have also been developed for robots with many degrees of freedom (6 or more) [68, 102, 111, 127]. But these planners still take too much time and/or lack consistency in their time performance to be used on-line.

Existing path planners can facilitate off-line robot programming and feasibility studies. For example, the path planner in [102] is used to compute collision-free paths of an 8-dof manipulator among cooling pipes in a nuclear plant. In [108] a planner generates paths of a 5-dof riveting machine to assemble portions of an airplane fuselage. The planner in [83] is used to check for the maintainability of aircraft engines.

Motion planning in the presence of obstacles moving along known trajectories is a step toward dealing with a dynamic environment. It has been studied in particular in [105, 106, 167, 172], where previous path planning methods have been extended to deal with the temporal aspect of this new problem. Motion plans are generated in the form of robot's *trajectories*, i.e., geometric paths indexed by time. The C-space is extended by adding a dimension, time, yielding the *configuration×time space*, or *CT-space*, of the robot. The obstacles map to a static forbidden region in this space. A trajectory is computed as a curve segment connecting the initial and goal configuration×time points and lying outside the forbidden region. This curve must be time-monotone, i.e., at any time t_0 its tangent must point into the half-space $t > t_0$. When the velocity of the robot is upper-bounded, the tangent must further point into a cone determined by the maximal velocity.

Motion planning for several robots sharing the same space has been addressed in [68, 69, 98, 112]. Two approaches have been proposed. The *centralized approach* consists of treating the various robots as if they were one single robot, by considering the Cartesian product of their individual C-spaces [68, 69]. This space is called the *composite C-space*. The forbidden region in this space is the set of all configurations where one robot intersects an obstacle or two robots intersect each other. A drawback of this approach is that it often leads to exploring a large-dimensional space, which may be too time-consuming. An alternative is the *decoupled approach*, which consists of planning for one robot at a time. In one technique, the robots whose motions have already been planned are treated as moving obstacles constraining the motions of the other robots [98]. This technique requires searching the C-space of the first robot and the CT-spaces of all other robots. Another technique, called velocity tuning, plans the path of each robot separately and then tunes the robots' velocities along their respective paths so that no two robots ever collide [112]. However, the decoupled approach is not complete, i.e., may fail to find a motion of the robots even if one exists.

Manipulation planning extends motion planning by allowing robots to move objects. It consists of interweaving transit paths, where a robot moves alone, and transfer paths, where it moves objects, separated by grasp/ungrasp/regrasp operations. These paths lie in different subspaces of the composite C-space defined as the Cartesian product of the C-spaces of all robots and movable objects. Manipulation planning has been studied for a single robot in [65, 179] and for multiple robots in [124, 125, 126, 127]. The regrasping issue with one robot has been specifically investigated in [178].

Contribution of this chapter: We believe that our planner is the first on-line planner able to solve complicated manipulation planning problems in a dynamic environment. This planner was implemented, connected to real robots, and applied to a practical scenario. Our experimental results are extremely satisfactory.

It is clear, however, that our planner utilizes a variety of techniques previously introduced in the literature surveyed above. It breaks the manipulation planning problem into a series of subproblems formulated in low-dimensional C-spaces; this idea was first proposed to effectively coordinate the motions of several robots. The planner solves each subproblem using a now classical best-first search algorithm guided by numerical goal-oriented potentials. It also draws on the concept of configuration×time space introduced to deal with moving obstacles and on the notions of transit/transfer paths (which approximately correspond to what we will call grasp/deliver paths) to deal with movable parts. Finally, it reuses efficient techniques to precompute obstacle regions in C-space as bitmaps enabling very fast collision checking.

On the other hand, our planner brings forward a series of new ideas, techniques, and experimental results. Its most important contribution is to demonstrate for the first time that complicated motion planning problems can be solved on-line to deal with dynamic environments. As many application domains may benefit from it, this contribution should motivate researchers to develop better on-line motion planning technology. The second main contribution of our work is precisely to provide a subset of that technology. Although we reuse known basic techniques, these are carefully re-designed and combined together to meet the challenging temporal constraints of a dynamic environment. The third important contribution is in the experimental evaluation. While off-line planners are evaluated by measuring their running times and characterizing the problems they can solve, evaluating an on-line planner is more complicated and subtle. We propose a series of measurable criteria, including the competitiveness of the planner relative to an instantaneous oracle, to characterize the planner's efficiency.

5.3 Scenario

The scenario involves two robot arms, a conveyor belt, an overhead vision system, a working table, movable parts, and obstacles. The arms must grab parts as they arrive on the belt and transfer them to specified goals on the table where, for example, they will form one or several assembled products.

The robot system is depicted in Fig. 2.1. It consists of two identical SCARA-type arms [89], each having three links and four degrees of freedom. The first two links of each arm form a horizontal linkage with two revolute joints. The third link, which carries the gripper, translates up and down. Finally, the gripper can rotate about its vertical axis. Each arm shares part of its workspace with the other arm, so that the same part may be grasped by one arm or the other; space sharing also allows for hand-over operations between the two arms. The arm that is closest to the beginning of the belt is called ARM1. The other arm is called ARM2; it can be seen as a backup for ARM1.

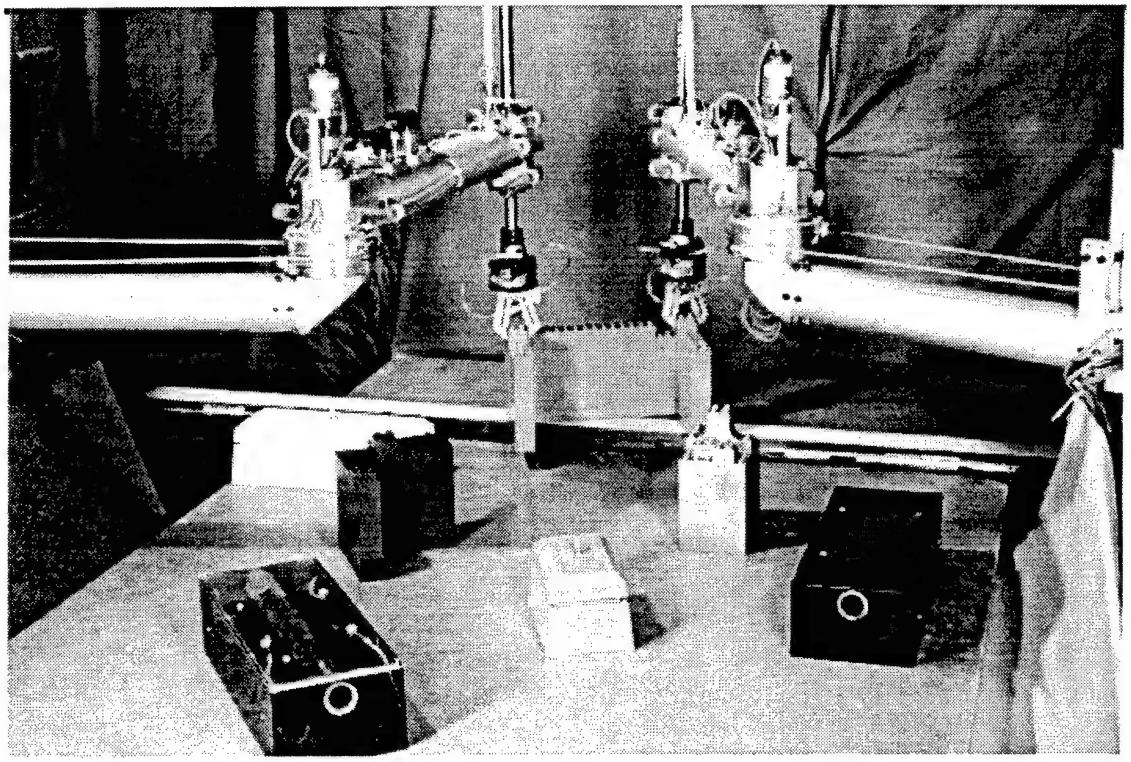


Figure 5.1: Two-arm robotic cell

Parts of different types arrive on the belt at any time, in random order, and with arbitrary positions and orientations. The vision system detects and identifies them, and tracks their locations while they are on the belt. The task of the arms is to grab as many parts as possible and transfer them to their goals (shown white in the figure), without collision. For each part X, the position and orientation (relative to X) where an arm's gripper can grasp X is unique and given. The goal of a part of any type is unique and within reach of at least one arm. When an arm releases a part at its goal, the part stays there until it is removed by an external mechanism. We assume that this mechanism never interferes with the arms, so that it is not taken into consideration by our planner.

Static obstacles (shown black in the figure), e.g., fixturing devices and other machines, are lying on the table. If an arm releases a part on the table, this part also becomes a static obstacle, until it is removed. All obstacles on the table lie below the horizontal volume swept out by the first two links of each arm. Similarly, an arm's gripper in its upmost position cannot collide with any obstacle. Hence, if an arm is not holding a part and its gripper is all the way up, it can only collide with the other arm. Such an arrangement is classical for SCARA-type arms, since otherwise motions would be too constrained to perform any useful task. However, when an arm holds a part, this part shares the same space as the obstacles. Hence, no part can be stacked on top of an obstacle or another part. But the belt is low enough so that when an arm holds a part with its gripper in its upmost position above the belt, the part is not hit by other arriving parts. This condition allows an arm

to lift a part above the belt and stay there for a while, e.g., waiting for the next motion command.

A single-processor computing resource is dedicated to planning. The planner, which gets all its information about the arriving parts from the vision system, must use this resource to decide on-line which arm motions to execute in order to transfer as many parts as possible to their goals. Ideally, the efficiency of the planner should be measured against an instantaneous oracle that would always make the best decision. The ratio N_p/N_o, where N_p (N_o) denotes the numbers of parts successfully moved to their goals when the planner (the oracle) is in command, computed over a run, defines the efficiency of the planner for that run. The closer to 1, the better.

Example: The above scenario is illustrated in Fig. ??, ??, and ??, with a series of snapshots produced by our planner. Snapshots are indexed by time; the run starts at time 0 and the sampling rate is 0.25sec/frame. Each snapshot displays two configurations of the arms and moving parts: the one in dark grey is the current configuration; the one in light grey is an intermediate configuration between the previous and the current snapshots. Parts of two types are fed during the run. We denote them by X_i and Y_j, where X and Y refer to the pentagonal and T-shaped parts, respectively, and i and j indicate the order of arrival. Each part disappears as soon as it is delivered to its goal.

In snapshots (2)-(4), ARM1 (the top arm) and ARM2 (the bottom arm) are simultaneously delivering X_2 and X_1 to their goals. In (6), X_1 reaches its goal and disappears. In (7)-(11) ARM1 performs the deliver motion of X_2, while ARM2 clears the way for this motion. There are two new parts, Y_1 and X_3, arriving on the belt. In (12), immediately after ARM1 has delivered X_1 to its goal, ARM2 starts executing a motion to grasp Y_1. Simultaneously, ARM1 performs a short motion to free the way for ARM2, as shown in snapshots (12)-(13). Snapshots (12)-(21) display the grasp motion of ARM2. Concurrently, in (15)-(21), ARM1 performs a motion to catch X_3. In (21), the arms grab X_3 and Y_1.

In (24)-(34) ARM2 delivers Y_1 to its goal, while ARM1 is staying still holding X_3 above the belt. In (36)-(38) ARM2 clears the way for ARM1, which starts executing the deliver motion of X_3. Note that the goal of X_i has changed between (34) and (36). This change is taken into account in ARM1's deliver motion, as shown in snapshots (40)-(51). In (40)-(42) ARM2 executes a grasp motion to catch Y_2 and starts moving Y_2 toward its goal in (51). In (53), since X_3's goal is not reachable by ARM1, ARM1 releases X_3 at an intermediate location reachable by ARM2. X_3 then becomes an additional obstacle which is taken into account by the deliver motion of ARM2 shown in (56)-(66). In (53)-(60) ARM1 first frees the way for ARM2 and then performs a grasp motion to catch Y_3. In (64)-(66) it starts transferring Y_3, while ARM2 is still delivering Y_2 to its goal. In (68) ARM2 clears the way for ARM1, which delivers Y_3 to its goal in (70)-(76). In (70)-(78) ARM2 changes posture before grasping X_3 in (80) and moving it toward its goal in (82)-(85).

In (70) part X_4 leaves the workspace ungrasped. In (78) ARM1 has finished delivering Y_3 and starts moving toward the belt to grasp Y_4. In (82)-(96), ARM1 grasps Y_4 and moves it to its goal. In (88)-(98), ARM2 moves to grasp X_5. Finally, in (98)-(100), ARM1 starts another grasp motion to catch part X_6.

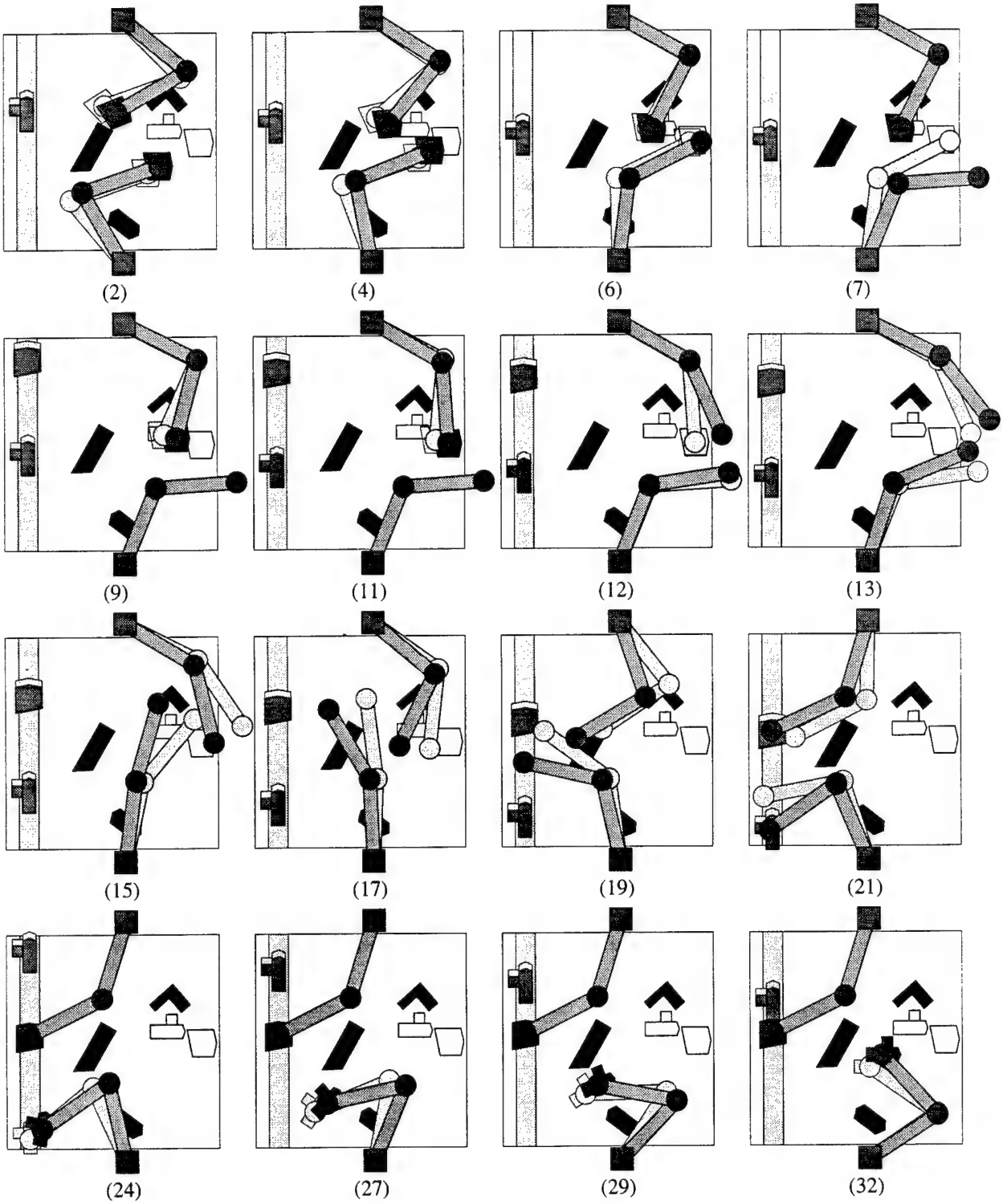


Figure 5.2: Example (part 1)

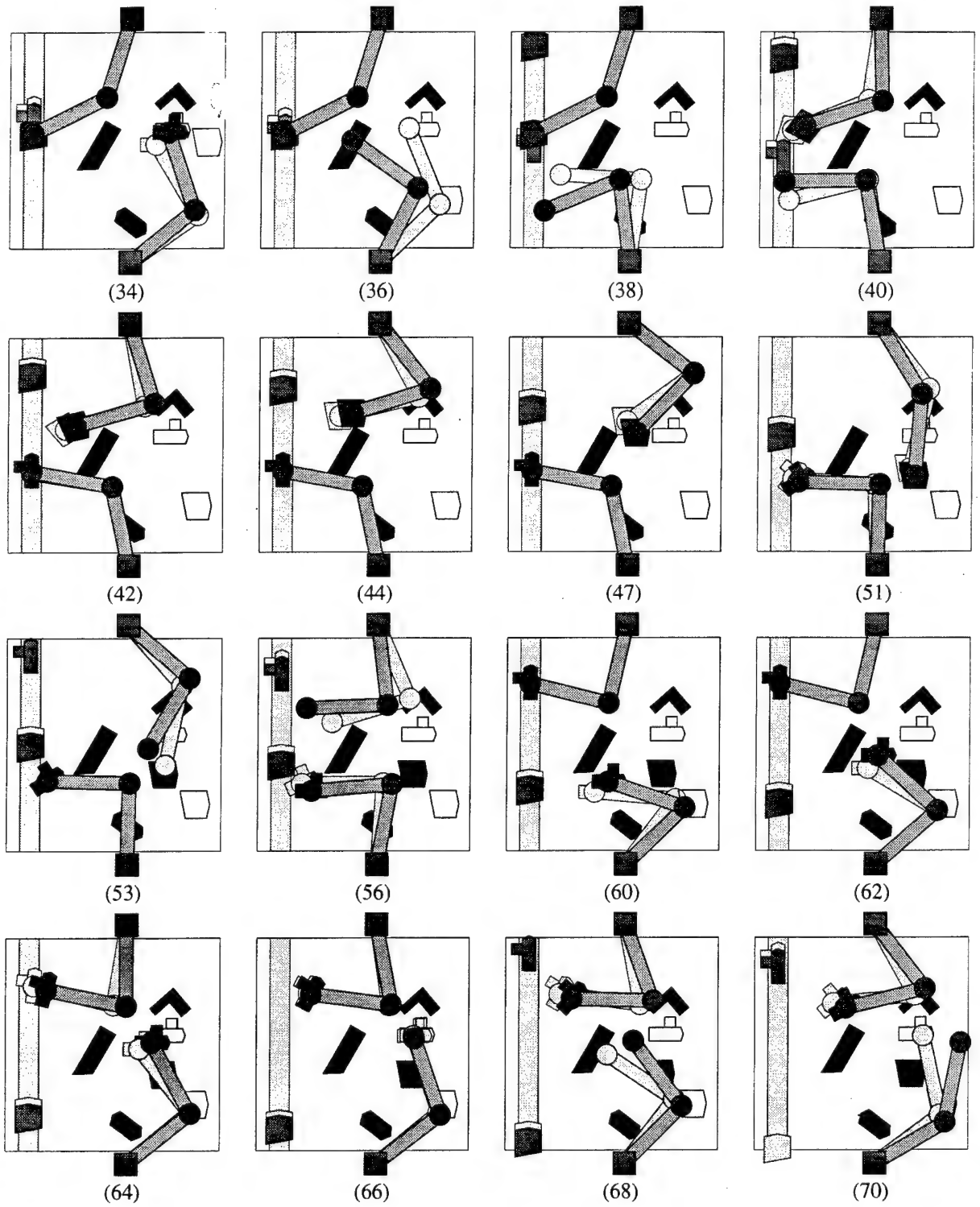


Figure 5.3: Example (part 2)

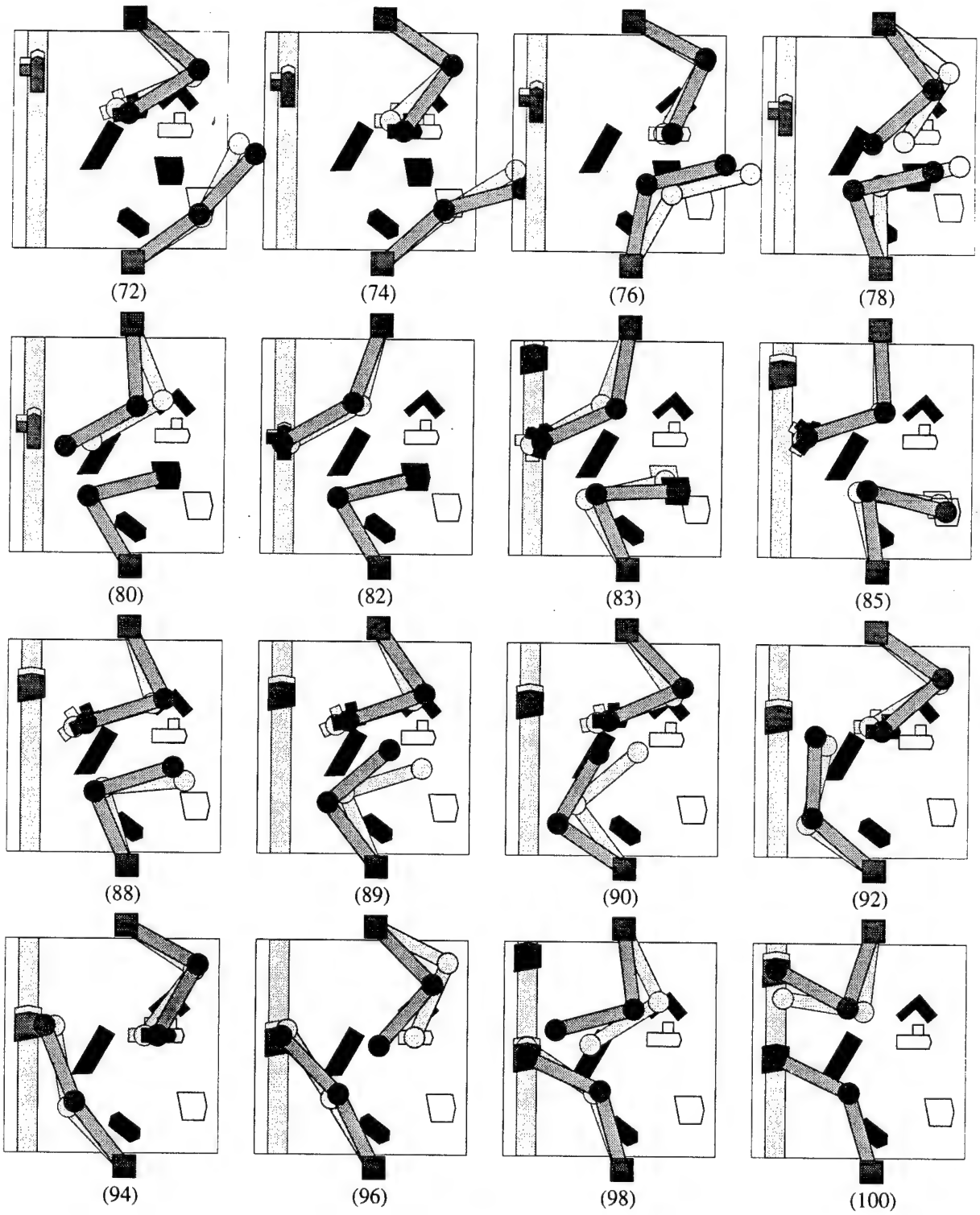


Figure 5.4: Example (part 3)

In Section ?? we will consider extensions of this scenario allowing for dynamic changes in the types of parts, the goals of the parts, and the obstacles. We will also address the case where some parts require being held by two arms simultaneously, in order to be moved.

5.4 Overview

In this section we outline our approach to on-line manipulation planning and we state the assumptions and heuristics it relies on. We denote the arms by A_1 and A_2, with A_1 standing for either ARM1 or ARM2, and A_2 standing for the other arm.

5.4.1 Planning Primitives

In principle, if the arrival times of the parts on the belt were known in advance, the manipulation planning problem of our scenario could be solved off-line by searching through the high-dimensional composite C-space of the two arms and the parts. However, the fact that manipulation is constrained by time, i.e., that parts must be grabbed at the right place at the right time, would still make the problem very tricky. For instance, one can imagine sequences of incoming parts where it is preferable to let a part pass ungrasped, though it could be grasped, because this will free the arms for grabbing more parts later. Today, there exists no planning method that would do the work, without taking prohibitive time to run. So, the problem needs to be simplified.

Obviously, performing planning on-line makes simplifications even more necessary. One way to proceed is to consider several low-dimensional C-spaces, rather than a single, large-dimensional one. However, such simplification yields a planner that is no longer complete. Usually, the more simplification, the faster, but the less complete the planner.

So: How much simplification is suitable? We use the following rule-of-thumb: Planning a motion should take significantly less time than executing this motion. We justify this rule as follows: If planning is longer, the performance of the robot system degrades quickly; but if it only takes a small fraction of the time needed for execution, making it even faster has little effect on the system efficiency. This justification is actually supported by experimental evidence with our implemented planner (see Section ??). The above rule leads us to reduce the planning problem to a succession of subproblems in spaces of dimension three. Indeed, there exist techniques that plan motions in such spaces much under the second on current workstations, while planning in spaces of dimension four or higher takes one or several orders of magnitude longer [68]. Reducing planning to even smaller spaces, if possible, would yield faster, but weaker primitives. The time gain would contribute little to the total efficiency of the robot system; but the greater weakness of the primitives would likely lead to missing opportunities.

Our planner mainly searches through two types of 3D spaces: the CT-space of an arm and the C-space of a part. We briefly discuss below the assumptions and heuristics which allow us to decompose the problem and limit planning to these spaces.

First, consider the arms. We allow the first two links of an arm to move only when its gripper is all the way up. Hence, an arm can only collide with the other arm. This allows us to represent the two arms in a 2D workspace as shown in Fig. ??-??. Links are modeled by rectangles and joints by disks. The disk at the extremity of each second link also includes the vertical link and the gripper. With this representation, planning a collision-free motion of the two arms requires dealing with a 4D C-space. To reduce the problem further, we decouple arm planning so that we always plan for a single arm at a time, say A_1, while the other arm, A_2, can be idling or executing a previously planned trajectory. This problem can be formulated as computing a trajectory in the 3D CT-space of A_1. A part X moving on the belt maps into A_1's CT-space as the set of all tuples (q_1^g, t_g) such that if A_1 is at configuration q_1^g at time t_g, it can grasp X.

To make our presentation shorter, we will assume that translating the gripper to grasp or ungrasp a part is instantaneous, and that orienting the gripper can always be coordinated with the motions of the other two links. We will also consider that parts disappear from the table immediately after they have been delivered to their goals. These assumptions are easy to remove and not made in the implemented planner.

Now consider a part X. We represent X as a 2D object by projecting it on the horizontal plane. When X is being transferred by an arm, the part both translates and rotates in the plane, hence tracing a path in its 3D C-space. Stationary obstacles map into this C-space as a forbidden region that X's path must not intersect. Our planner computes X's path between its grasp and goal configurations in the subset of its C-space that is reachable by at least one arm. Through the arm's inverse kinematics, this path then entails the path of the arm holding X. If X leaves the space reachable by this arm, the planner will command the arm to ungrasp X at an intermediate location on the table where it can be regrasped by the other arm.

One way to make it possible to plan the path of X in its 3D C-space is to require that the arms move no more than one part at a time. But such a condition may lead one arm that has picked up a part to wait for the completion of the other arm's motion before actually moving the part. On the other hand, planning for the motion of two parts X and Y simultaneously entails reasoning in their 6D composite C-space and planning for one part at a time, considering the other part as a moving obstacle, still requires searching a 4D CT-space. This leads us to use the following technique: The planner always computes the path of a part X in its 3D C-space. If another part Y is currently being moved, this motion is temporarily ignored. When X's motion has been computed, its execution is coordinated with the current motion of Y by computing the earliest time when X's motion can start without causing collision.

In addition to the simplifications made above, our planner assumes perfect sensing. It also assumes that the arms can perfectly track the planned trajectories, with each joint being able to instantaneously change velocity. Unlike previous simplifications, these assumptions yield discrepancies between the planning model and the real world. In Section ?? we will discuss how we overcome these discrepancies in order to run the planner with real robots.

5.4.2 Planning Processes

A crucial issue in on-line motion planning is to react to events by focusing quickly on urgent subproblems and sizing opportunities to grab parts before they vanish. *Planning processes* are the main tool used by our planner to manage its activities over time.

Whenever a new part is detected on the belt by the vision system, a message is recorded in a queue Q. When a part is grabbed or reaches the end of the belt, the corresponding message is removed from Q. At any time, Q lists all the parts currently on the belt or at intermediate locations on the table, along with their current configurations.

Assume that the planner starts with no parts on the belt and the table, and no arms moving. The arrival of a part on the belt triggers planning which consists of selecting a part X in Q (here, there is no choice, but usually there is one) and an arm A_1, and setting up a planning process whose task is (X, A_1), i.e., plan a motion of A_1 to grasp and deliver X to its goal. This process will be terminated upon the completion or the failure of its task. Although its main task is to plan for A_1, this process may also generate a motion of A_2; for instance, if A_2 is currently immobile or if its motion ends before the yet-to-be-planned motion of A_1 is over, the process may command A_2 to free the way for A_1. We call such a motion an *accommodating motion.*

A new process is created whenever a process is killed or interrupts itself to allow for the execution of an already planned motion (process interruption will be presented in the next subsection), and there exist a part $X \in Q$ and a non-moving arm A_i ($i = 1$ or 2) such that neither are currently assigned to a planning process. (The planner considers that an arm is moving as soon as it has sent a motion command to the robot controller, and that it has stopped moving when it is told so by the controller.) Since there are only two arms and each process "consumes" one arm, no more than two processes can exist simultaneously.

The robot operations in our scenario may be accomplished with different orderings of the parts and different assignments of arms to parts. Computing plans for all possible orderings/assignments and choosing the one that can grasp the largest number of parts in the shortest time would take prohibitive time to run. This would even not guarantee to produce the best solution, since this solution may also depend on parts yet to come. Instead, we assign tasks to processes according to the following heuristic rules:

H1: The parts at intermediate locations on the table have higher priorities than those on the belt, since they may obstruct possible paths for these new parts.

H2: The parts that are more advanced on the belt have higher priorities than those which are less advanced, since they will leave the arms' workspace earlier.

H3: When both arms are not moving, ARM1 has higher priority than ARM2, since ARM2 is at the ending side of the belt and thus can be used as a backup for ARM1.

A planning process with task (X, A_1) may fail to solve its task; for example, it may not find a trajectory for A_1 to grab X from the belt on time. The process is then killed. According to the

third rule above, if A_1 was ARM1, then there is still a chance that the process $(X, \text{ARM2})$ will be created. For every part in the queue Q, the planner keeps track of failures to avoid reassigning a combination part-arm to a new process, if that combination is guaranteed to fail again.

5.4.3 Concurrent Planning

Recall from Section 2.3 that a single processor is always available for planning. While a planning process is using this processor for a task (X, A_1), other parts may arrive on the belt and the other arm A_2 may be idling. The planner could wait until the current process is killed before creating a new process. However, it may be urgent to plan for A_2, in order to avoid missing parts. But the task of the first process is also urgent. This conflict leads us to break the task (X, A_1) performed by a process into two subtasks: the *grasp subtask* – plan a trajectory to grasp X – and the *deliver subtask* – plan a trajectory to deliver X to its goal.

A grasp subtask, in general, is more urgent than a deliver subtask since the latter's goal is time-independent. For that reason, a planning process interrupts and puts itself on hold between the two subtasks, allowing for the creation of a new process. More precisely, suppose that a process P_1 is created to plan for the task (X, A_1). P_1 first plans the motion for the grasp subtask and then puts itself on hold if a feasible motion has been found. While A_1 is executing this motion, the processor is free and can be used by other processes, say P_2, to plan for other tasks. Once A_1 has grasped X, P_1 resumes and solves for the deliver subtask. However, if P_2 is currently running, P_1 is put in the waiting state until P_2 is either interrupted or killed. While waiting for a deliver path, A_1 stays still with its gripper holding X all the way up above the belt, so that parts arriving on the belt can pass below. Processes thus take turns in using the processor.

Hence, a planning process P may traverse the following states during its lifetime:
- *Running*: P is running if it uses the computing resource to compute a plan.
- *On hold*: P is on hold while the arm assigned to it performs the grasp or deliver motion.
- *Waiting*: P is waiting if it needs to compute a delivery motion, but the computing resource is being used by another process.

At any one time, there exist 0, 1, or 2 planning processes. In theory, while one process P is on hold or waiting, several other processes can be successively created (and killed). In practice, the number of these other processes is small; indeed, as soon as one solves its grasping subtask or no new parts arrive on the belt, P will be running again.

If a process whose task is (X, A_1) fails to solve either the grasp or deliver subtask, it is immediately killed. However, in the second case, A_1 is already holding X. Then A_1 releases X on the belt as soon as there is enough distance between two incoming parts. (We will see that if X was picked up from the table, the delivery subtask cannot fail.) Part X is updated in the queue Q accordingly. There might still be a chance that the other arm can accomplish the task.

A planning process P_1 may be unable to plan the motion of an arm A_1 to deliver a part X at its goal because this goal lies outside the space reachable by A_1 or because obstacles force X's path

to leave A_1's reachable space. Then P_1 does not fail. It produces a motion of A_1 that delivers X at an intermediate configuration where it can be regrasped by the other arm A_2. Q is updated, P_1 is killed, and it will require another process P_2 (with arm A_2) to move X to its goal. Rule H1 in Subsection ?? will give this part a higher priority than any other part on the belt. This kind of hand-over operation from one arm to the other can happen more than once for the same part.

The interruption of a planning process between the grasp and deliver subtasks increases planning efficiency only if the deliver subtask rarely fails. In a realistic robot setting, the obstacles on the table should be distributed to allow incoming parts to be moved from the belt to their goals. But it may occur that, due to hand-over operations, all paths for a part are obstructed by other parts resting at intermediate locations on the table and waiting to be regrasped.

5.4.4 Deadlock Analysis

Can the planner come to a deadlock? Two types of deadlocks may be thought of: computational and physical.

A computational deadlock corresponds to the case where a process waits indefinitely for a resource to free. However, each of the processes created by the planner is a stand-alone computation that receives all its data at the time it is created and this computation is always finite (this will become clearer in the next section where we describe the planning primitives in detail). Hence, our planner is free of computational deadlocks.

A physical deadlock corresponds to the case where the decisions made by the planner yield a physical situations where no more motions can be performed. This could happen after the arms have released several parts at intermediate locations on the table. Since no such part is ever transferred back to the belt, they could obstruct their respective paths to the goals as well as the paths of the new arriving parts, hence creating a physical deadlock. This situation never occurs for the following reason: Whenever a part X is transferred to an intermediate location, the planner has already found a complete path for X to its goal (if the planner had failed, it would have released X on the belt). If several parts lie simultaneously at intermediate locations on the table and the planner decides to transfer one to its goal (the result of applying heuristic rule H1), it gives the highest priority to the most recent part (see Section ?? (A) for more detail). This part is guaranteed to have a path to its goal. If the only arm that can perform this path is not available, the planner will have to consider another part (the next most recent) on the table. But, eventually, if all parts except the most recent one have their paths obstructed, the planner will create a deliver task for this most recent part.

The above argument is correct only if the obstacles and the goals remain unchanged. This condition will no longer be true in Section ??. Then physical deadlocks will become possible. They are not treated in the current planner.

In any case, the absence of deadlocks, although a desirable property, says little about the planner's efficiency. For example, the planner may still be too slow to make it possible for the arms to grasp

any part arriving on the belt. Many successive processes would then be created, but they would all fail to produce grasp trajectories. Also, although the planner does not have to move parts lying on the table to the belt in order to avoid physical deadlocks (when goals and obstacles are fixed), one could certainly imagine circumstances where such an operation would increase efficiency. Only comprehensive experiments can give us a reasonable measure of the planner's efficiency. Hence, the importance of Sections ?? and ??.

5.5 Planning Techniques

We now present a detailed account of the activities carried out by the planner within the lifetime of a planning process solving for the task (X, A_1).

5.5.1 Representation of C-Spaces and CT-Spaces

Each arm A_i, $i = 1$ or 2, is modeled as a planar two-revolute-joint linkage; hence, it has a 2D C-space C_i. We parameterize a configuration q_i of A_i by the arm's two joint angles, θ_{i1} and θ_{i2}. Each angle spans an interval of amplitude less than 2π determined by mechanical stops.

We use the following metric over C_i: Let $\omega_{i,1}$ and $\omega_{i,2}$ be the respective maximal velocities of the first and second joints of arm A_i. The distance $D(q_i, q_i')$ between two configurations $q_i = (\theta_{i1}, \theta_{i2})$ and $q_i' = (\theta_{i1}', \theta_{i2}')$ of A_i is:

$$D(q_i, q_i') = \max\{\frac{|\theta_{i1} - \theta_{i1}'|}{\omega_{i,1}}, \frac{|\theta_{i2} - \theta_{i2}'|}{\omega_{i,2}}\}.$$

This definition is consistent with our assumption that arm joints achieve their planned velocities instantaneously: $D(q_i, q_i')$ then measures the minimal time that A_i takes to travel between q_i and q_i'. The straight-line segment joining q_i and q_i' in C_i is the shortest among all possible paths connecting these two configurations; its length is $D(q_i, q_i')$.

The CT-space CT_i of A_i is defined as $C_i \times [0, +\infty)$, with C_i parameterized as above and the third dimension being time. At every point $(\theta_{i1}, \theta_{i2}, t)$ in CT_i, the maximum velocities $\omega_{i,1}$ and $\omega_{i,2}$ define a cone of points reachable from $(\theta_{i1}, \theta_{i2}, t)$.

Each part X arriving on the conveyor belt has a 3D C-space. A configuration of X is parameterized by the coordinates x and y of a reference point attached to X in a fixed coordinate system and the angle θ defining the orientation of X relative to this system. The pair (x, y) is called the *position* of X. While an arm is holding X, the arm's configuration determines the position of X.

All C-spaces and CT-spaces searched by our planner are represented as bitmaps. Cells containing "1"s designate the forbidden region where collision occurs. Cells containing "0"s form the free region in which paths and trajectories must lie. We will describe efficient techniques to construct

these bitmaps in Subsection ??. The set of cells in a CT-space bitmap projecting onto the same time interval is called a *time slice.*

The resolution along the two axes of an arm's C-space bitmap is chosen so that, when the arm is fully extended, moving the first joint or the second joint by one increment roughly causes the same displacement ε of the arm's endpoint. The resolution along the time axis of the CT-space bitmap is such that when one joint moves at maximal velocity during an increment of time, the arm's endpoint moves by approximately one increment in the C-space bitmap. The resolution of the C-space bitmap of a part X is the same along the two dimensions representing X's position and roughly equal to ε. The resolution along the orientation axis is such that when X rotates by one increment about its reference point, the point of X that undergoes the maximal displacement (i.e., the point in X the farthest away from the reference point) moves by approximately ε [68].

5.5.2 Planning a Grasp Motion for a Part on the Belt

Let us now consider the grasp subtask of (X, A_1). X may either be a part arriving on the belt, or a part previously ungrasped at an intermediate location on the table. The two cases receive basically the same treatment, with a few minor differences. Here we consider the first case, which is also the most frequent.

We let t_c stand for the current time. Hence, t_c is continuously changing value. We let q_1^s designate the configuration of A_1 when it starts executing the grasp motion.

For any given time t, we let $q_1^g(t)$ denote the configuration of A_1 at which it can grasp X. The map q_1^g is defined over the time interval $[t_g^{min}, t_g^{max}]$ during which X is on the belt within A_1's reach. It may yield two configurations, since the inverse kinematic equations of A_1 usually have two distinct solutions corresponding to two postures of the arm. The planner always selects the configuration which is closest to q_1^s according to the metric D. This is a reasonable choice since this grasp configuration is likely to be the quickest to reach. (An alternative would be to successively apply the planning techniques presented below to both configurations defined by the map q_1^g and select the configuration allowing for the earliest grasp. This variant would roughly take twice as much planning time, but could produce a more efficient motion plan once in a while.)

We distinguish between two cases: In (A), the second arm A_2 is not moving; in (B), it is moving.

(A) Arm A_2 is not moving: While a grasp motion is being planned and then executed, X keeps moving on the belt. To plan A_1's motion, we must know when and where X can be grasped, which in turn depends on how much time it will take to plan and execute this motion. This difficulty yields the following iterative procedure.

We initially schedule the grasp at the latest possible time, i.e., we set $t_g = t_g^{max}$. *If A_2 is not holding a part,* the planner generates A_1's path between q_1^s and $q_1^g(t_g)$ as the straight-line segment connecting these two points in C_1. An accommodating motion of A_2, if needed, is generated in CT_2,

into which A_1's path maps as a forbidden region. Since actual velocities will be computed later, we momentarily take time equal to the abscissa along A_1's path. A_2's motion is computed in the form of a time-monotone curve, i.e., a curve whose tangent at any time t_0 points into the half-space $t > t_0$. This curve joins $(q_2^s, 0)$ to some (q_2^e, L), where q_2^s denotes the current configuration of A_2. q_2^e stands for any configuration of A_2 where it does not collide with A_1 at the grasp configuration $q_1^g(t_g)$, and $L = D(q_1^s, q_1^g(t_g))$.

If the line segment joining $(q_2^s, 0)$ and (q_2^s, L) in the bitmap representing CT_2 traverses "0" cells only, no accommodating motion of A_2 is needed. Otherwise the accommodating trajectory is constructed by searching the CT_2's bitmap in a depth-first manner for a sequence of free cells connecting the cell containing $(q_2^s, 0)$ to the time slice containing L. At every iteration of the search, let c be the last cell reached (initially, c is the cell containing $(q_2^s, 0)$). The algorithm considers the seventeen cells adjacent to c in the same or next time slice, and moves to a closest free cell that is not already in the current path (the distance between two cells being measured as the distance D between the projections of their centers into C_2). This greedy algorithm tries at each step to minimize the time that will be required to execute the accommodating trajectory, but by no means does this guarantee an optimal result. If all of the seventeen cells are non-free or already part of the current path, the search backtracks and possibly fails.

If A_2 is holding a part Y, the planner could proceed as above. But it would also have to make sure that Y does not hit any obstacle on the table during the accommodating motion of A_2. This additional check would yield longer computation time. For this reason, the planner does not make A_2 accommodate to A_1's motion. Instead, it treats A_2 as a static obstacle and searches C_1 for a path of A_1 avoiding this obstacle.

At this stage, the planner knows the geometry of the coordinated paths of A_1 and A_2 (possibly, A_2's path is void). It then computes the velocity profiles of the joints to execute these paths in minimal time. This is done by discretizing the curvilinear abscissa along A_1's path into small intervals and, in each interval, letting the joint that takes the longest time at maximal velocity set the pace for the other joints. This computation yields the duration δ of the coordinated motion, hence the latest starting time $t_s = t_g - \delta$ for the motion. If t_s is smaller than the current time t_c, grasping X with A_1 is considered impossible and the planning process is killed; the pair (X, A_1) will not be reassigned to a planning process. Otherwise, the grasp scheduled at t_g is feasible. But if $t_s - t_c$ is rather large, say, more than a few times the time spent planning the motion, executing this motion will lead A_1 to wait for the arrival of X. A better motion may then be possible and the planner iterates the above procedure by scheduling an earlier grasp time t_g. At the end of every iteration, the latest starting time of the best motion computed so far is used to bound the computation time allowed to the next iteration. If one iteration exceeds the time allocated to it, it is aborted and the best motion computed so far is executed. The successive times t_g are chosen by dichotomically decomposing the interval $[t_g^{min}, t_g^{max}]$.

A motion computed as above is shown in snapshots (12)-(21) of Fig. ??. In this example, A_1 is ARM2 moving to grasp Y_1; in snapshots (12)-(13), ARM1 performs a short accommodating motion that clears the way for ARM2. (The grasp motion of ARM1 shown in snapshots (15)-(21) is generated

by the technique described in Case (B) below.)

(B) Arm A_2 is moving: The motion being performed by A_2 constrains the future motion of A_1. It is mapped to a forbidden region in CT_1 and the trajectory of A_1 is computed between some (q_1^s, t_s) and some $(q_1^g(t_g), t_g)$, where $t_s > t_c$ stands for the starting time of A_1's motion and $t_g \in [t_g^{min}, t_g^{max}]$ is the time when A_1 grasps X.

Let us temporarily assume that A_2's motion is scheduled to end after time t_g^{max}. Therefore, if A_1 can grasp X, its motion will terminate before the one of A_2. As in case (A), the planner iteratively determines t_g. For every t_g such that A_1 at $q_1^g(t_g)$ does not obstruct A_2's trajectory at any time $t \geq t_g$, it searches for a trajectory joining the line $\{(q_1^s, t) | t > t_c\}$ to $(q_1^g(t_g), t_g)$, avoiding the forbidden region, satisfying the joint velocity constraints, and starting after t_c.

The planner performs the search backward, from the selected $(q_1^g(t_g), t_g)$ toward the line $\{(q_1^s, t) | t > t_c\}$, using a best-first technique. At every iteration of the search, it selects a pending node (q_1, t) of the current search tree such that $D(q_1^s, q_1)$ is minimum over all pending nodes. It computes nine potential successors of this node by successively setting the velocity of each joint to zero, its maximal value with positive sign, and its maximal value with negative sign, and integrating the corresponding motion over the duration of a time slice in the bitmap representing CT_1. If a potential successor belongs to a "0" cell c of this bitmap and c has not been visited before, then it is included in the search tree as a new pending node and c is marked 'visited'. The actual point is recorded in the search graph, so that the velocity bounds along the entire trajectory are perfectly respected. The best-first algorithm and the definition of D guarantee that the computed trajectory takes minimal time over all valid trajectories in the discretized search space. The search fails when no leaves in the search tree lie in time slices occurring after t_c.

The planner selects the successive values of t_g in increasing order between t_g^{min} and t_g^{max}, at the centers of the time slices in CT_1's bitmap. If the search fails for one value of t_g and another value is considered, the new search will discard every node's successor lying in a cell visited by a previous search. Indeed, at the bitmap resolution, this successor cannot be on a valid trajectory, otherwise the previous search would not have failed. Therefore, each new value of t_g usually yields a small amount of additional computation, and the total number of search nodes generated to compute A_1's trajectory is at most equal to the number of free cells in CT_1's bitmap. At soon as the planner succeeds in finding a trajectory for A_1, this motion is executed.

Let us now consider the case where A_2's motion is scheduled to end at time $t_2 < t_g^{max}$. Beyond t_2, A_2 may then perform another motion to clear the way for A_1. To take advantage of this possibility, the planner proceeds as follows: For every selected value of t_g greater than t_2, it generates A_1's grasp trajectory as the concatenation of two trajectories: one connects q_1^s to some q_1^i chosen as the closest configuration to $q_1^g(t_g)$, outside the forbidden region at time t_2; the other connects q_1^i to $q_1^g(t_g)$ and may require an accommodating motion of A_2. The second motion is computed first (if $q_1^i \neq q_1^g(t_g)$) using the techniques of case (A), with q_1^i substituted for q_1^s and q_2^s replaced by A_2's expected configuration when it terminates its current motion. If the latest starting time t_s' of the computed motion is less than t_2, the motion cannot be performed, and the planner tries the next

value of t_g. Otherwise, a trajectory of A_1 between $\{(q_1^s, t) | t_c < t < t'_s\}$, and (q_1^i, t'_s) is generated using the above backward best-first search algorithm, with A_2 mapping into the same forbidden region in all the time slices of CT_1 between t_2 and t'_s. If this computation yields a starting time greater than t_c, the motion is executed; otherwise the next value of t_g is considered. Again, the marking of the cells of CT_1's bitmap saves considerable time.

An example of a motion computed as above is the motion of ARM1 to grasp X_3 in (15)-(21) of Fig. ??. This motion terminates approximately at the same time as the ongoing grasp motion of ARM2 and requires no accommodating motion of ARM2. Another example is the motion of ARM2 to catch Y_2 in (40)-(42) of Fig. ??; this short grasp motion ends before the ongoing transfer motion of ARM1 terminates. A third example is the motion of ARM1 to grasp Y_3 in (53)-(60). A fourth example is the motion of ARM2 to grab X_5 in (88)-(98); this motion ends after the ongoing deliver motion of ARM1, but does not require ARM1 to perform an accommodating motion. A fifth example is the motion of ARM1 to grasp Y_4 in (78)-(82) of Fig. ??. This motion required no accommodating motions of ARM2. A sixth example is the motion of ARM1 to grasp X_6 in (96)-(100).

5.5.3 Planning a Grasp Motion for a Part on the Table

In this case, X is not moving; hence, time is slightly less critical. We let q_1^g denote the configuration of A_1 where it can grasp X; if two such configurations are feasible, we select the one that is closest to q_1^s. Again, we distinguish between cases: (A) A_2 is not moving; (B) it is moving.

(A) Arm A_2 is not moving: This case could be treated like case (A) in the previous subsection, with one difference: there is no need for guessing successive values of the grasp time. However, we proceed in a slightly different manner. Whether A_2 is holding a part or not, we first treat it as a static obstacle and we try to generate a path of A_1 avoiding this obstacle. If such a path is found, it is executed. Otherwise, if A_2 is not holding a part, we plan a motion of A_1 along a straight-line path and we make A_2 accommodate to this motion.

The first tactic, which treats A_2 as a static obstacle, aims at avoiding an accommodating motion of A_2, since while executing such a motion, A_2 cannot grab a new part on the belt. However, this tactic tends to fail more often than the second one, which makes A_2 accommodate. Since X is not moving, we can afford to waste a short amount of computation time trying the less reliable tactic first and using the other tactic as a backup.

(B) Arm A_2 is moving: The treatment is as in case (B) of the previous section, with t_g chosen at the centers of the successive time slices beyond the current time. Since X does not move, one iteration eventually succeeds.

In snapshots (68)-(80) of Fig. ??-??, the motion of ARM2 to grasp X_3 on the table illustrates this case. This motion changes ARM2's posture, because grasping X_3 with the other posture is not feasible.

5.5.4 Planning a Deliver Motion

Like in the previous subsection, the goal of the motion is time-independent; but now we must plan for both X and A_1.

(A) Arm A_2 is not moving: The planner first generates a path connecting the initial and goal configurations of X by conducting a best-first search in the bitmap representing X's C-space. This search is guided by a goal-oriented potential field similar to the NF2 function described in [130] and is restricted to the configurations of X where A_1 and/or A_2 can grasp X, as proposed in [127]. An alternative, which could save time-consuming hand-over operations, is to first restrict the search to the configurations of X where A_1 can grasp X; only if this search fails, the configurations reachable by A_2 would also be considered. The generation of a path for X may fail due to parts previously placed on the table by the arms. Then A_1 puts X down on the belt or the table at its current location and the planning process is killed. The planner will not reassign the task to grasp X to any arm as long as none of the parts currently on the table has been removed.

If a path is found for X, it entails a path for A_1 through the arm's inverse kinematics. The initial posture of A_1 is the one at the end of the previous grasp path. If in this posture one joint of A_1 reaches a limit, while changing posture would allow A_1 to continue transferring X, the planner includes an ungrasp operation in A_1's path before the joint limit is attained, then a subpath to change A_1's posture, and finally a regrasp operation, before resuming tracking X's path. A_1 may change posture several times along X's path. If A_2 lies along the way of A_1's path, a motion of A_2 to clear the way for this path is generated. As much as possible, we would like to avoid a long accommodating motion of A_2, since during this motion A_2 cannot grasp new parts. The planner proceeds as follows:

- It first tries to generate a path of A_2 to clear the way for A_1. This motion is planned in C_2, into which the discrete sequence of configurations describing A_1's path (minus the subpaths changing A_1's posture) maps as a union of forbidden regions. The final configuration of A_2's is any configuration outside this union. If a path is found, A_2's final configuration is mapped to a forbidden region in C_1 and the subpaths to change A_1's posture are computed then. The path of A_2 is executed at maximal velocity. The motion of A_1, also at maximal velocity, starts as soon as it can no longer collide with A_2. To determine the starting time of A_1's motion, the planner maps A_2's trajectory to a forbidden region in CT_1. It then represent A_1's trajectory as a curve segment in CT_1 with its initial point at the time when A_2 is scheduled to terminate its motion. Finally it translates this curve toward smaller values of time. The position of the curve just before it intersects the forbidden region due to A_2 gives the starting time of A_1's motion.

- If the previous computation fails to generate paths for A_1 or A_2, the planner computes an accommodating motion of A_2 as in case (A) of Subsection **??**. Prior to this computation, it completes the path of A_1 by inserting the subpaths changing the arm's posture. These subpaths are simply straight-line segments in C_1.

If X's path leaves A_1's workspace, the planner commands A_1 to put down X at an intermediate

configuration where it can be regrasped by A_2 and the planning process is killed. The planner memorizes that A_2 will have to be assigned to the regrasp of X. Note that when A_1 releases X, there exists a path for X to its goal. But this path may later be obstructed by additional parts ungrasped on the table. For that reason, the planner computes regrasp motions for parts on the table by reversing the chronological order in which they have been ungrasped. In this way, there is no need to recompute paths for these parts.

In snapshots (24)-(34) of Fig. ??-?? the deliver motion of ARM2 to transfer Y_1 to its goal was computed as above. The path computed for Y_1 directly entailed a path of ARM2 free of collision with ARM1. Snapshots (36)-(51) illustrate the above planning techniques in a more complicated case, in which ARM1 must move X_3 to its goal. This goal was changed between (34) and (36) and is now out of reach of ARM1. The path of X_3 entails a path of ARM1 that collides with ARM2. Thus, a motion of ARM2 is planned to clear the way and is executed in (36)-(38). It is followed by the motion of ARM1 in (40)-(51). The path of X_3 is then close to leaving the space reachable by ARM1; so, in (51), ARM1 ungrasps X_3.

(B) Arm A_2 is moving: The planner generates X's path as in case (A), with one difference: If A_2 is currently transferring a part to an intermediate configuration, the part at this configuration is treated as an additional obstacle for X. The path of X entails a path of A_1. A_1's motion at maximal velocity along this path is coordinated with A_2's current motion using the same technique as above, that is, by translating A_1's trajectory toward smaller values of time. However, if A_2 holds a part Y, collision must be avoided not only between A_1 and A_2, but also between X and Y. This is done by using a separate bitmap representing the C-space of X relative to Y. This check guarantees that if X's goal lies along Y's path, X will not be moved to an obstructing location before Y has already been through that location.

There is an additional difficulty. It may happen that the final configuration of A_2 lies along A_1's path. Then let t_2 denote the time when A_2's motion is expected to terminate. A_1's motion is coordinated with A_2's motion by mapping A_2 into CT_1 until time t_2 only. Thus, A_1 will execute as much as possible of its motion prior to t_2. At t_2, the motion of A_2 ends and the motion of A_1 is also temporarily stopped. We then plan a motion of A_2 to clear the way for A_1 as in case (A). Though the planning process for A_1 is idling between the time A_1 starts moving and t_2, it is not put on hold; since A_2 is moving, this arm could not be assigned to another part anyway.

The above computation is illustrated with the motion of ARM2 in snapshots (51)-(66) of Fig. ??. This motion was planned and is executed while ARM1 is still moving. Since ARM1 is going to release X_3 at an intermediate location, this part will become an additional obstacle that is taken into account by the planner when it computes the path of Y_2 (see (60)-(64)). Here the final configuration of ARM1 lies along the way of ARM2, which requires planning a motion of ARM1 to clear the way. As indicated above, the motion of ARM2 is temporarily stopped (see (53)). The new motion of ARM1 is executed in (53)-(56); it precedes ARM1's grasp motion in (56)-(60). A second deliver motion (arm ARM1) computed as above is shown in (64)-(76). Because ARM2 obstructs the path of ARM1, a motion of ARM2 to clear the way is first planned and executed in (68). Notice

that immediately after, starting in (70), ARM2 starts executing a grasp motion to grab the part X_3 lying on the table. This motion was computed after the motion of (68) was executed; the planner then realized that it was safe to execute it concurrently with the ongoing motion of ARM1. A third illustration of the above computation is the motion of ARM1 shown in (82)-(96).

5.5.5 Bitmap Construction

The role of a bitmap representing a C- or CT-space is twofold. It provides a discretization of a continuous space prior to searching that space. It also allows for quasi-instantaneous collision checks. Of course, we must consider the cost of generating the bitmap, but most of this computation can be done in a preprocessing phase. Furthermore, computing an entire bitmap is often not more time-consuming than performing a few explicit collision checks in the workspace. The idea of using precomputed bitmaps to accelerate collision checking is also used in [127, 131].

Part's C-space: The C-space bitmap for a part X represents the forbidden region created by the obstacles. We model both X and the obstacles as unions of convex polygons, $\{X_i\}_{i=1,2,...}$ and $\{O_j\}_{j=1,2,...}$, respectively. Every pair (X_i, O_j) of convex polygons yields a subset of the forbidden region in X's C-space. Any cross-section of this subset at a constant orientation of X is itself a convex polygon that is computed in time linear in the number of vertices of X_i and O_j [109, 144]. A polygon-filling function transforms this polygon into a 2D bitmap. The 3D C-space bitmap of X is constructed by stacking fixed-orientation 2D slices. Each slice is generated by computing the forbidden regions due to all pairs (X_i, O_j) and drawing them into the same 2D bitmap.

If all obstacles were fixed, the C-space bitmap for a given type of part would only be computed once. However, the obstacles include parts that have been temporarily released on the table. In the next section we will also allow dynamic changes in the obstacles. Whenever there is a change in the obstacles, the bitmap must be updated. To reduce updating costs, we precompute a bitmap representing the forbidden region corresponding to every pair part-obstacle and part-part. The size of this bitmap is just large enough to enclose the forbidden region; hence, it is much smaller than the full C-space bitmap. The C-space bitmap is first computed by filling it with "0"s and copying the "1"s of each individual bitmap in the right cells. Whenever an object is removed from the table, the bitmap is recomputed in the same way. When a new object is placed on the table, "1"s are added to the C-space bitmap according to the individual bitmaps involving this object.

In case (B) of Subsection ??, we allow a part X to move while another part Y is already moving. This requires performing collision checks at various configurations of X and Y. The part-part bitmap corresponding to the types of X and Y is used then. Whenever a collision check is needed, the relative configuration of X and Y is computed; this configuration determines the bitmap cell to look into.

Arm's C-space: A technique to compute the 2D C-space bitmap of an arm A_1, given the configuration of the other arm A_2, is to enumerate all configurations in the bitmap (e.g., the centers of the cells) and, for each one, test if it is collision-free. Collision checking between arms requires considering four or three pairs of links, depending on whether the first links of the two arms can touch each other, or not (in our setting, only three pairs of links must be considered). To check if two links collide, we simply look into a link-link bitmap. This bitmap is precomputed using the technique of the previous paragraph by treating one link as a fictitious robot free to translate and rotate in the plane and the other link as an obstacle. This technique is reasonably fast, but it can be improved as follows.

In each arm, we choose the reference point of the second link at the center of rotation of the second joint. Thus, if we fix the first joint angle θ_{11} of arm A_1, the reference point of the second link is also fixed. Given the configuration of A_2, we scan all possible values of θ_{11} in C_1's bitmap. Each value determines a cell in two bitmaps, each representing the interaction between the first link of A_1 and a link of A_2. If the first link of A_1 collides with a link of A_2, the whole column in C_1's bitmap is filled with "1"s. Otherwise, the position of the reference point defined by the current value of θ_{11} determines a column in the bitmaps representing the interaction between A_1's second link and each of the two links of A_2. After shifting these two columns appropriately (to align their origins with the origin of the column of C_1's bitmap at the current θ_{11}) and removing the cells beyond the second-joint mechanical stops, we compute their boolean union and copy the result into the column of C_1's bitmap at the current value of θ_{11}. In our implementation this improvement cut the time to compute an arm C-space bitmap by almost two orders of magnitude.

Other efficient techniques to compute C-space bitmaps for articulated arms are proposed in [74, 147, 151].

Arm's CT-space: The 3D bitmap representing an arm's CT-space is computed one time slice at a time. The computation of the 2D bitmap in one time slice is done as above, only when the planner needs this time slice. A 2D bitmap is memorized at least as long as it is beyond the current time.

5.6 Extensions and Improvements

Changes in goals: The goal of a part can be changed at any time. If a part X arrived before its goal changed, but the deliver motion has not been computed yet, the new goal will be used by the planner when it solves for the deliver subtask. If, instead, the deliver motion has already been planned, it is executed without modification. However, immediately after X reaches its previous goal, the planner plans a new motion to transfer it to its new goal. Similarly, if X has been delivered to its goal and this goal changes prior to the removal of X by the external mechanism, the planner generates a motion to transfer X to the new goal.

One possible exploitation of this planner's ability is the following: Let the parts be used to assemble

several copies of a product. Parts delivered to their goals are removed only when a product is complete. However, since parts arrive in random order, too many parts of one type may arrive during a period of time. Because the goals of these parts are occupied, the arms will let them pass ungrasped. Instead, one can define several sites for assembling the product. Initially, the goal of each part is in the first site. When this goal is filled, a new goal is set in the second site, and so on. Whenever a site contains a complete product, this product is removed and all the goals in that site become free again.

New types of parts: New types of parts can be dynamically introduced. For the user, adding a new part means describing its geometry, defining its goal, and specifying the grasp position of a gripper. For the planner, it only requires computing new bitmaps. As long as the number of obstacles and parts of different types is not too large, this computation can be carried out on-line without significantly weakening the total system performance.

Changes in obstacles: The positions and orientations of the obstacles can be changed. However, we impose that such changes happen when no deliver motion is being executed. They only require the planner to update the C-space bitmaps of the incoming parts. This modification is very fast.

Obstacles can also be added or removed. Whenever an obstacle is added, new bitmaps describing the interaction of this obstacle with the various types of parts that may be fed are computed.

Cooperative manipulation: We allow the introduction of parts that require two arms to be moved, say, because they are too heavy for a single arm, or because they have elongated shapes. Two grasp positions are defined for the grippers on each such part.

Let us assume that such a part X arrives on the belt. The planner assigns it to a planning process only if the two arms are non-moving. The two arms are also jointly assigned to this process. The planner generates the grasp motion very much like in case (A) of Subsection ?? by iteratively guessing a grasping time t_g. However, at each iteration it must compute the coordinated motion of the two arms: First, it generates the path of ARM1 to attain the grasp position on X (at t_g) that is closest to the beginning of the belt. This path is simply constructed as a straight-line segment in ARM1's C-space. Next, the planner generates a coordinated path for ARM2 to attain the other grasp position on X (at t_g). This motion is computed by searching through ARM2's CT-space, with time taken equal to the abscissa along ARM1's path. Planning for the deliver subtask is done like in Subsection ??. No hand-over operations are possible, but the planner may put down the part on the table to change an arm posture or swap grasps (see [126]).

Anticipating catches: The process coordination described in Section ?? may sometimes let the planner idling. In fact, rather than idling, if an arm is not moving, the planner then generates a motion for that arm to bring it to a predefined configuration where its gripper is close to the belt. Thus, when a new part arrives on the belt, the arm will be in a better position to catch it quickly.

We could extend this idea and keep the planner always busy by making it work on what may happen beyond the next round of motions. To be really fruitful, however, this generalization requires additional study.

5.7 Implementation

The planner described in Sections ??, ??, and ?? has been fully implemented in C on a DEC Alpha workstation (Model Flamingo) running under DEC OSF/1. This machine is rated at 126.0 SPECfp92 and 74.3 SPECint92 on the SPECMARKS benchmark. The planner has been connected to both a robot graphic simulator and a real robotic system.

The robot system in our simulator has the same general characteristics as the real system. The lengths of the first and second links of each arm are both 24in. The first joint rotates within a 135dg interval and the second in a 285dg interval. The maximal velocities of the joints are 15.2dg/sec. The belt moves at 4in/sec. The interval of time during which a part can be grasped is approximately 15sec.

An arm C-space bitmap has size 36×76, which corresponds to increments along each axis of about 3.75dg. The size of the bitmap representing a part C-space is on the order of $128 \times 128 \times 96$; the increments along each of the two position axes are approximately 0.57in long. Having the same increments along all angular axes allows for simplifications in the planner's code; nevertheless, setting the size of the increments as suggested in Subsection ?? raises no particular difficulty and should be done in a new version of the planner.

We performed various tests with our software to measure the running times of some of its key components. We obtained the following average times for a representative sample of components:

- Computing a bitmap representing the interaction between two objects takes 41ms.[1]
- Computing a complete C-space bitmap for a new type of part using the precomputed part-obstacle and part-part bitmaps takes 8.3ms.
- Updating a part C-space bitmap when an object is added onto the table takes 5.6ms.
- Constructing an arm C-space bitmap using the precomputed link-link bitmaps takes 0.4ms.
- Searching a part C-space bitmap with the best-first search technique of Subsection ?? is done at a rate of 65,000 nodes/sec.
- Searching an arm CT-space with the best-first search technique used in case (B) of Subsection ?? is done at a rate on 870,000 nodes/sec.

The sequence of snapshots shown in Fig. ??-?? was produced by our planner connected to the graphic simulator.

[1] Our implementation uses no hardware-implemented polygon-filling function to compute such a bitmap.

Feeding Uncertainty (sec)	0.0	0.5	0.5	0.75	0.75	1.0	1.0
Slowing Down Feeding Rate	No	No	Yes	No	Yes	No	Yes
Number of Parts per Minute	13.19	13.19	11.88	13.19	11.32	13.19	10.81
Missing Ratio (Oracle)	0%	20%	0%	25%	0%	28%	0%
Missing Ratio (On-line Planner)	13%	19%	6%	17%	2%	14%	0%

Table 5.1: Comparison of our on-line planner to a quasi-optimal oracle

5.8 Evaluation in Simulated Environment

We have generated several measures of the efficiency of the planner connected to the graphic simulator, by running it on multiple sequences of arriving parts. Each run lasts 8min, during which on the order of 100 parts are being fed. No parts require being moved by two arms simultaneously. Parts disappear immediately after they are delivered to their goals, so that no part is ungrasped because its goal is occupied. The simulator uses the same model of the physical world as the planner. We define the *missing ratio* of the planner over a run as the number of parts that go ungrasped, in percents of the total number of parts fed during this run.

Comparison to quasi-optimal oracle: Ideally, the planner's efficiency should be evaluated relative to an instantaneous oracle always making the best decision. However, building such an oracle is not realistic, since it requires implementing an optimal off-line manipulation planner. Instead, we built a quasi-optimal oracle as follows: We let each of the two arms move at maximal velocity along a simple path connecting two configurations, one where the gripper is above the belt, the other where it is above the table away from the belt. The two arms perform these motions alternately, forward and backward, so that when one arm is above the belt the other arm is at the other end of its trajectory above the table. The trajectories are defined so that no collision occurs in the middle. Then we define a feeding sequence of parts so that when an arm reaches the end of its trajectory above the belt, a part is right there to be grasped and the goal of this part is exactly at the other end of the arm's trajectory. Finally, we distribute the obstacles on the table so that no part collides with an obstacle when it is moved by an arm. By construction, the missing ratio of this oracle for the sequence of parts defined above is 0%.

We ran the planner with the same obstacle distribution and the same sequence of parts. Table ?? compares results obtained with the oracle and the planner. In column 1, we feed the part with no uncertainty. The number of parts fed per minute is 13.19. The missing ratio of the oracle is 0%, while the missing ratio of the planner is 13%.

In columns 2 and 3 of the table, instead of feeding parts at exactly the times computed above, we let them arrive within a ± 0.5sec uncertainty interval. If the feeding rate is unchanged (column 2), the missing ratio of the oracle increases sharply to 20% (this is obtained by temporarily stopping the arms' motions whenever an arm reaches the belt prior to the arrival of the part); on the other hand,

CPU speed factor	S_1	S_2	S_3	S_4	S_5	average
0.1	34	33	29	28	35	31.8
0.2	26	30	23	22	27	25.6
0.5	18	15	15	13	12	14.6
1.0	9	10	9	13	8	9.8
2.0	6	6	10	9	10	8.2

Table 5.2: Effect of planning time on missing ratio

the missing ratio of the planner increases slightly to 19%. Let us slow down the feeding rate just enough so that the oracle's missing ratio becomes zero again (column 3); this requires stopping the arm motions given by the oracle, for a maximum of 1sec prior to any grasping operations. The belt now feeds 11.88 parts/min. The missing ratio of the planner is also reduced to 6%. The subsequent columns show similar results when feeding uncertainty is $\pm$0.75sec and $\pm$1sec. When the feeding rate is slowed down just enough to make the oracle's missing ratio equal to 0%, the planner's missing ratio drops to 2% and 0%, respectively. Note the stability of planner's performance throughout these experiments.

We have run several experiments similar to the above. In all cases, our planner showed the same high degree of competitiveness. Combined with the fact that it also allows for dynamic changes in the parts and the obstacles, these results suggest that in many situations an on-line planner such as ours can be more attractive than an excellent off-line planner.

Effect of planning time: We also analyzed the effect of planning time by artificially changing the planner's running speed. This is done as follows: Whenever the planner solves for a grasp or deliver subtask, we interrupt the simulator and measures the planner's running time. When the computation is over, we let the simulator update the state of the environment according to the running time of the planner. For this update, we can set the running time as we desire, e.g., to twice what it actually was, or half of it, or even zero. The variations of the missing ratios for different planning speeds gives us an idea of the planner's efficiency if we used a slower or faster computer.

Table 2 gives the missing ratios of the planner for five feeding sequences $S_1, \ldots, S_5$ and five planner's speeds (0.1, 0.2, 0.5, 1.0, and 2.0 times its nominal speed). When the planner's speed is half the nominal one, the missing ratio is still reasonably small. When the speed is twice the nominal one, the planner's performance is not greatly improved. When the planner's speed is even greater (not shown in the table), planning time becomes negligible relative to execution time, and the missing ratios remain approximately constant.

These results suggest that, as computers become faster, it will be worth making the planner devote more computation than it currently does generating motion plans that are quicker to execute, e.g.,

belt speed	S_1	S_2	S_3	S_4	S_5	average
0.5	15	10	20	20	13	15.6
0.75	12	5	9	15	11	10.4
1.0	9	11	9	13	8	10.0
1.25	7	8	8	13	12	9.6
1.5	10	11	7	11	11	10.0
2.0	6	11	12	11	12	14.4
2.5	19	16	16	20	16	17.4
3.0	20	21	20	25	19	21.0

Table 5.3: Effect of belt velocity on missing ratio

by reducing the number of hand-over operations and the number of changes in arm posture, and by increasing parallelism between motions.

Effect of belt velocity: Table 3 shows the planner's missing ratios for five different runs and eight different velocities of the belt (0.5, 0.75, 1.0, 1.25, 1.5, 2.0, 2.5, and 3.0 times the nominal velocity). The feeding rate is fixed, so that slowing down the belt results in more parts on the belt at any one time. In general, the table indicates that the planner's performance is rather insensitive to the belt speed. However, above some speed (about twice the nominal speed), it degrades more rapidly. When the belt is slowed down, one could expect an increase in performance, since parts stay longer on the belt and so can be grasped over larger intervals of time. Surprisingly, the contrary happens. This seems to be caused by the presence of more parts on the belt at the same time, leading the arms to constrain each other more severely than when the speed is higher. This suggests that our heuristics (rule H3 in Subsection ??) to assign parts to arms should be improved in this case; e.g., we may too frequently assign ARM1 to an arriving part whose goal is not reachable by ARM1 or hard to reach, while we could let this part advance on the belt and then assign it to ARM2. This kind of improvement could benefit the planner even when the belt's velocity is nominal.

5.9 Connection to Robotic System

System description: We have connected our planner with the dual-arm robotic system shown in Figure ??, which has been developed in the Aerospace Robotics Laboratory at Stanford [165]. The integrated system comprises five major modules: the user interface, the on-line manipulation planner, the dual-arm robot control system, the real-time vision system, and the graphic simulator [160, 161]. Characteristic arrangements of LEDs are mounted on all objects of interest (arriving parts and obstacles); the overhead vision system senses these LEDs, identifies their arrangements, and computes the positions of the objects in real time. This information leads to updating a model of the environment that is used by all other modules. For example, the planner learns that a new

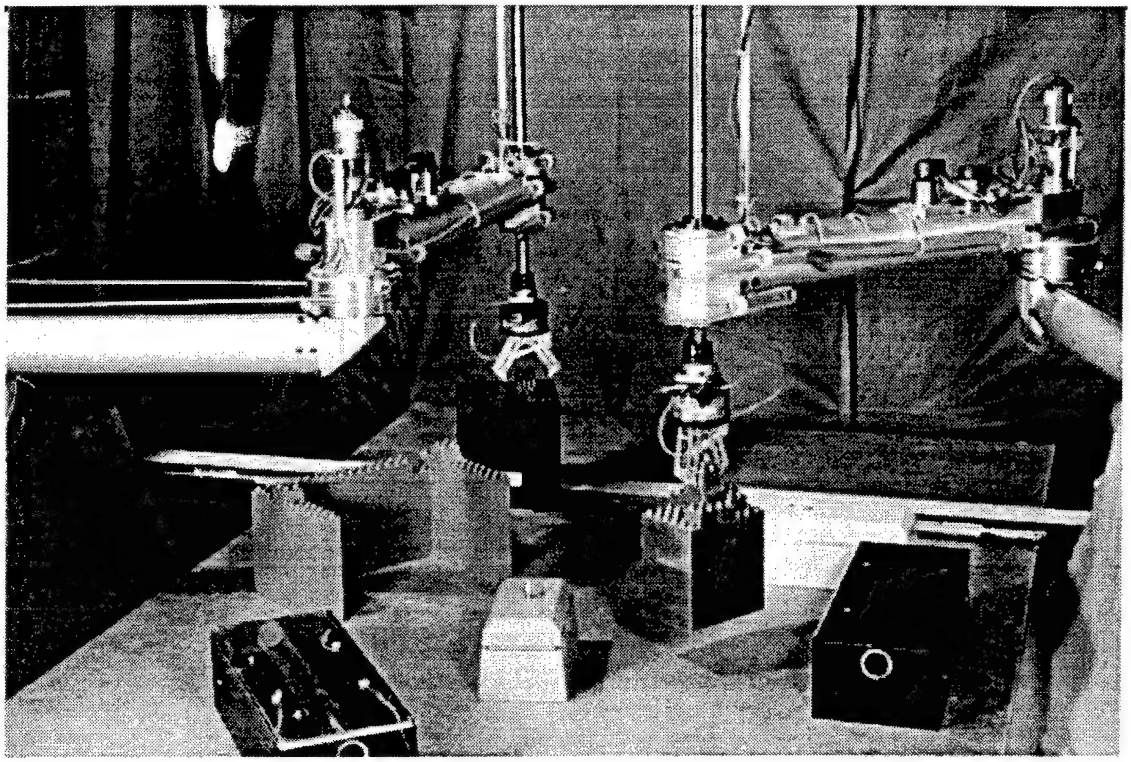


Figure 5.5: Dual-arm robotic system and experimental setup

part arrives on the belt or that the position of an obstacle has changed by periodically accessing this model. The user interface provides commands to interactively specify and modify the goals of the parts arriving on the belt. The graphic simulator allows the user to observe graphic renderings of what the vision and the control modules believe is going on in the real world and what the planner predicts will happen soon.

This software is integrated under ControlShell, which provides object-oriented tools for combining software components developed separately [63]. The five modules are implemented on several computers and communicate through a subscription-based network data sharing system called the Network Data Delivery Service (NDDS) [162]. In the current implementation, the user interface and the planner run on two different UNIX workstations, while the control and vision modules run on several VME-based real-time processors.

We have successfully experimented with this integrated system on various examples similar to the one shown in Fig. ??-??, as well as on examples involving long objects requiring two arms for their transfer.

Interfacing the planner with the rest of the system: The planner assumes that the arm joints can change velocity instantaneously. Planned trajectories are thus impossible to execute

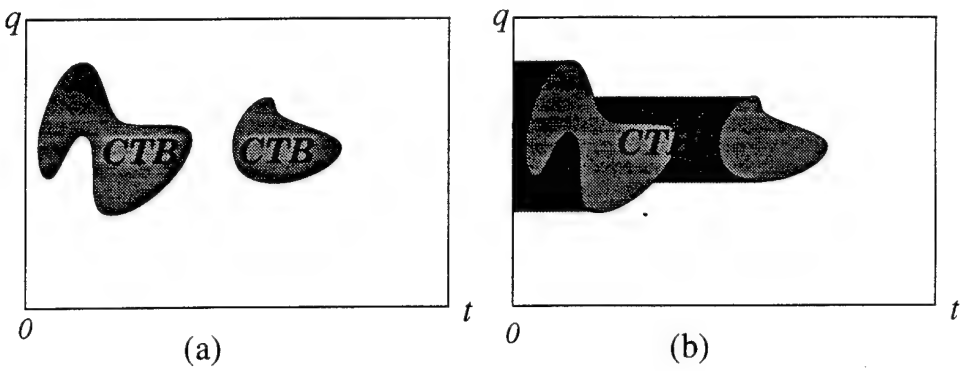


Figure 5.6: Extending the forbidden region in CT-space

accurately. Hence, after a trajectory has been passed by the planner to the controller, the latter recomputes its time parameterization using a realistic dynamic model of the arms. The new trajectory has the same geometry as the one produced by the planner.

Let us assume for a moment that the arms can exactly execute the recomputed trajectories. In some cases, to guarantee that the trajectories recomputed by the controller are still collision-free, the planner takes a more conservative planning approach than the one previously described. For example, consider a grasp trajectory of A_1 to be performed while A_2 is moving. The planner maps A_2's trajectory to a forbidden region in CT_1 and extends this region by its shadow along the negative time dimension, as illustrated in Fig. ??. (Note that the previous generation of A_2's motion accounted for the current configuration of A_1; therefore, this configuration lies outside the extended forbidden region.) Thus, the planned trajectory of A_1 can be arbitrarily translated toward the right (greater values of time) without causing any collision. It is then sufficient for the controller, when it recomputes A_1's trajectory, to make sure that A_1 is never ahead of time relative to A_2.

With slight modifications as the above, recomputed motions remain collision-free. But an arm may now fail to arrive in time to grab a part on the belt. This problem is handled by setting the maximal joint velocities in the planner smaller than the actual values. Some tuning is necessary: If the velocity values selected for the planner are too small, the planner will often fail to find grasp paths; but if the values are too large, the actual motions will often arrive too late to grasp the parts. On our implementation, the velocity bounds given to the planner are constants that have been estimated through preliminary experiments. Using these bounds, it may happen (though rarely) that an arm arrives too late to grasp a part. In that case, the planner may decide to make another try by generating another motion, possibly with the other arm.

In our network-distributed implementation, the planning and control modules run on different machines. Communication delays are not negligible. We model them by a duration proportional to the length of the transmitted trajectory, plus some small latency. The planner adds this delay to the computed start time of a grasp trajectory to determine if the robot reaches the grasp configuration

on time.

We assumed above that the arms perfectly track the recomputed trajectories. However, control errors cannot be totally avoided. In our implementation we bounds the angular errors of the joint angles of an arm by two constants η_1 and η_2. These bounds specify that, if the arm is expected to be at configuration q at time t, it may actually be anywhere in the parallelepiped centered at q whose sides have lengths η_1 and η_2. Every configuration along a trajectory of A_1 is mapped into CT_2 by considering the region swept by A_1 between the two extreme configurations it may achieve. Appropriate link-link bitmaps are precomputed, so that this computation brings no additional cost.

Vision sensing is also imperfect. Errors in the positions of the objects on the table measured by the vision system have been bounded by preliminary experiments. These bounds are used to grow the geometric models of the objects used by the planner. Errors on the grasp position of a part are also bounded in order to generate safe C-space bitmaps for the parts.

Finally, we must consider the grasping operations on the belt. Taking conservative approaches as above would not allow an arm to reliably grasp a moving part. Hence, we proceed differently: When a gripper arrives within some distance to the part it is expected to grasp, the controller does not try to track further the planned trajectory. Instead, it tracks the part using the last position given by the vision sensor (when the gripper is almost above the part, this sensor no longer sees the part) and the measured velocity of the conveyor belt (which makes it possible to infer the motion of the part). When the gripper is above the part, both moving at the same velocity, the controller commands the grasp operation. While in this autonomous mode, the controller checks for collision between the arms. If one is going to happen, it stops both arms. It also reports the failure to the planner, which may try to generate new trajectories to catch the parts that have been missed. During a grasp operation, the vision sensor keeps updating the environment model. By accessing this model, the planner is informed of the new configurations taken by the grasping arm.

Evaluation of planner: The way we deal with discrepancies between the world model used in planning and the real world affects the efficiency of the total system. Thus, the following question arise: Could it be preferable to use a more realistic model of the real world at planning time? Using such a model would certainly increase planning times, but on the other hand it would also reduce delays in the interface between the planner and the robot controller. We do not have a definite answer to this question, but we have conducted experiments that shed some light on it.

Our experiments consisted of running the planner connected to a modified version of the simulator used in the previous section. Consider a motion generated by the planner. Let δ be the duration of the motion according to the planner's model. The modified simulator executes this motion in a time randomly selected in the interval $[\delta \times (1.0 + \mu - \eta), \delta \times (1.0 + \mu + \eta)]$, where μ and η are two parameters. Table 4 gives the planner's missing ratio for the first feeding sequence used in Table 2, for several values of μ and η. The third entry of the first row in Table 4 is the missing ratio (9%) when there are no discrepancy between the planner and simulator models.

Note that the missing ratio increases more rapidly when $\mu > 0$ (i.e., when the planner over-estimates

	$\mu = -0.5$	$\mu = -0.25$	$\mu = 0.0$	$\mu = 0.25$	$\mu = 0.5$
0.0	12	13	9	17	29
0.[illegible]	11	10	10	16	31
0.5	13	11	14	18	23
average	12	11.33	11	17	27.67

Table 5.4: Effect of belt velocity on the missing ratio of different part sequences

the performance of the arms) than when $\mu < 0$ (the planner under-estimates the performance of the arms). When the planner over-estimates performance, the motions arrive late to grasp the parts; this requires the arms to spend more time tracking the parts on the belt; the chances to miss parts also increase. When the planner is conservative, it may fail to generate grasp motions that would actually be feasible; but whenever a grasp motion is found, this motion is usually executed with success; moreover, less time is wasted tracking the parts to be grasped. Table 4 also shows that the planner is rather insensitive to the variations of η. By comparing Table 4 and the first column of Table 2, we notice that the effect of $\mu = 0.25$ has about the same magnitude as doubling planning time, while setting $\mu = 0.5$ has about the same effect as increasing planning time by a factor of 4.

These results combined with those of Section ?? (Effect of planning time) suggest that future increase in computer speed could usefully be exploited by making use of a more realistic arm model at planning time.

5.10 Conclusion

This chapter has described an on-line manipulation planner for a dual-arm robot system whose task is to grab parts arriving on a conveyor belt and deliver them at specified goals. Parts arrive at any time, in random order. The planner uses information provided by a vision system to break the overall planning problem into a stream of rather simple subproblems and orchestrate fast planning primitives solving these subproblems. Experiments conducted with this planner in a simulated robot environment show that it compares very well to quasi-optimal oracles. Experiments with a real dual-arm robot system have demonstrated the viability of on-line planning in the real world. Since the planner also allows dynamic changes in obstacles, goals, and tasks, this result suggests that on-line planning may rapidly become more attractive than off-line planning (whose efficiency is also very sensitive to feeding accuracy). In fact, we believe that our on-line planner enables low-cost, flexible, and efficient part feeding.

Evaluation of the planner shows that, as computers become faster, future research should focus on spending more planning computation to produce motion plans that are quicker to execute (e.g., avoiding hand-over operations and changes in arm posture) and on using more realistic arm models to reduce delays in the planner/controller interface. Additional research could also be done to keep

the planner always busy, by making it anticipate future motions whenever there is time available for that. Other interesting research topics include dealing with more than two arms and with more than one computing resource (possibly shared with other activities, like sensing and control).

Chapter 6

Multi-Arm Manipulation Planning in Three-Dimensional Workspace

This chapter is based on the paper "On Multi-Arm Manipulation Planning," by Y. Koga and J.C. Latombe, published in Proc. of IEEE Int. Conf. on Robotics and Automation, *San Diego, May 1994, pp. 945-952.*

6.0.1 Introduction

Consider the task of programming a *single* robot arm to open a water faucet, and suppose two full revolutions of the handle are required. The robot must move and grasp the handle along a collision-free path. Then it must rotate the handle in the required direction. During this rotation, one joint of the arm will reach its limit, consequently requiring the robot to ungrasp the handle and regrasp it such that again it can be rotated. This process must be repeated until the two full revolutions are completed. Finally, the robot should ungrasp the handle and move back to some home position. Manipulation planning is the automatic generation of such a sequence of motion paths that delivers one or several movable objects (the faucet handle in this example) to a given goal configuration. This sequence is called a manipulation path. A crucial aspect of manipulation planning, relative to more classical path planning, is that it must account for the robot's ability to change its grasp of an object.

Manipulation planning becomes even more challenging when the robot system contains *multiple* arms. This brings the additional difficulty of deciding which arm(s), at any one time, must grab and move a movable object and how the various arms should cooperate along the manipulation path to ensure the delivery of the object to its goal configuration. Regrasping the object may now involve changing the arms grasping it. We call this the multi-arm manipulation planning problem.

There is much motivation for the study of automatic multi-arm manipulation planning. Multi-arm systems can be significantly more efficient than single-arm systems, e.g., by performing several motions simultaneously. They can also accomplish a greater variety of tasks. For example, in addition to the arms working independently of each other, they can cooperate to manipulate heavy and/or bulky objects by sharing the load for fast and responsive motions. One can also increase the workspace of the movable objects by having the robots pass the object from one arm to another.

In this paper we propose an implemented approach for solving the multi-arm manipulation planning problem. We illustrate our approach with a robot system consisting of three identical arms, each with six revolute degrees of freedom, operating in a 3D workspace. This system is similar to an experimental setup of three PUMA 560 arms operating in our laboratory. We restrict ourselves to the case where there is a single movable object in the environment, but we allow two types for this object: for one type, it requires two arms to move (e.g., the object is heavy); for the other type, it can be moved by a single arm at any one time. A task is specified by the description of the workspace (geometry of the obstacles, movable object, and arms), the initial and goal configurations of the movable object and arms, and the type of the movable object. The implemented planner computes a collision-free manipulation path to deliver the object to its goal configuration. It allows for the robots to change their grasp of the object, but, in its current version, it is not able to compute grasps. A set of potential grasps must be given as input, and the planner chooses among them.

Fig. 1 shows a series of snapshots along a manipulation path computed by our planner. The movable object is L-shaped and requires two arms to move.

Section 2 relates our work to previous work in manipulation planning. Section 3 gives a formal presentation of the multi-arm manipulation problem. Section 4 describes our planning approach; it introduces some simplifications to make the problem more tractable. Section 5 presents results obtained with the implemented planner.

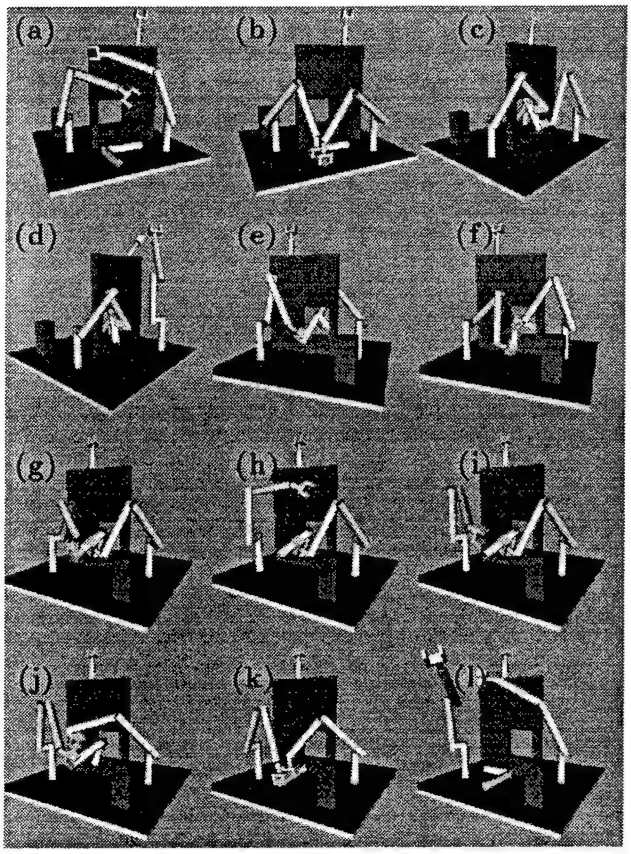


Figure 1. A multi-arm manipulation path. The object requires two arms to move it, but only one arm to hold it statically.

6.0.2 Related Work

Path planning for one robot among fixed obstacles and various extensions of this basic problem have been actively studied during the past two decades [130]. However, research strictly addressing manipulation planning is fairly recent.

The first paper to tackle this problem is by Wilfong [179]. It considers a single-body robot translating in a 2D workspace with multiple movable objects. The robot, movable objects, and obstacles are modeled as convex polygons. The robot "grasps" an object by making one of its edges coincide with an edge of the object. This definition of "grasping" extends to several movable objects. Wilfong shows that planning a manipulation path to bring the movable objects to their specified goal locations is PSPACE-hard. When there is a single movable object, he proposes a complete algorithm that runs in $O(n^3 \log^2 n)$ time, where n is the total number of vertices of all the objects in the environment. Laumond and Alami [134] propose an $O(n^4)$ algorithm to solve a similar problem

where the robot and the movable object are both discs and the obstacles are polygonal.

Alami, Siméon and Laumond [65] describe a manipulation planner for one robot and several movable objects. Both the number of legal grasps of each object (positions of the robot relative to the object) and the number of legal placements of the movable objects are finite. The method was implemented for two simple robots: a translating polygon [65] and a three-revolute-joint planar arm [135]. A theoretical study of the more general case where the set of legal grasps and placements of the movable objects are manifolds (continuous sets) is presented in [135].

Our work differs from this previous research in several ways. Rather than dealing with a single robot, we consider the case of multiple cooperating robot arms moving in a 3D workspace. In addition, whereas the previous work is more theoretical in nature, our focus is more on developing an effective approach to solve manipulation planning problems of a complexity comparable to that of the tasks encountered in manufacturing (e.g., assembling, welding and/or riveting the body of a car or the fuselage of a plane) and construction work (e.g., assembling truss structures). Similarly, in [?], Ferbach and Barraquand introduce a practical approach to this manipulation planning problem using the method of variational dynamic programming.

Along a slightly different line of research, Lynch addresses the problem of planning pushing paths [?]. He establishes the conditions under which the contact between the robot and the movable object is stable, given the friction coefficients and the center of friction between the movable object and its supporting surface. These conditions yield nonholonomic constraints on the the motion of the robot.

Regrasping is a vital component in manipulation tasks. Tournassoud, Lozano-Pérez, and Mazer [178] specifically address this problem. They describe a method for planning a sequence of regrasp operations by a single arm to change an initial grasp into a goal grasp. At every regrasp, the object is temporarily placed on a horizontal table in a stable position selected by the planner. We too need to plan regrasp operations. However, the only regrasps we consider avoids contact between the object and the environment; they necessarily involve multiple arms.

The work on regrasping presented in [178] is part of an integrated manipulation system, HANDEY, described in [146]. This system controls a single PUMA arm which builds an assembly in a 3D workspace. It integrates vision, path planning, grasp planning, and motion control. While it embeds a solution to many issues not considered in this paper, it does not address the problem of planning cooperative robot motions to accomplish manipulation tasks.

Planning coordinated paths for multiple robots, without movable objects, is studied in several papers, e.g. see [?].

Grasp planning is potentially an important component of manipulation planning. In our planner, grasps are selected from a finite predefined set of grasps. A better solution for the future will be to include the automatic computation of grasps. A substantial amount of research has been done on this topic. See [163] for a commented list of bibliographical references.

We have done some prior work in manipulation planning. In [126] we propose several planners to

generate manipulation paths for two identical arms in a 2D workspace. In [125] we extend one of these planners to allow the manipulation of several movable objects; this problem raises the additional difficulty of selecting the order in which the objects should be moved. Experiments have been conducted with this planner using a real dual-arm robot system developed in the Aerospace Robotics Laboratory at Stanford University [161].

Research in multi-arm manipulation planning is made more critical by the advent of new effective techniques to control cooperative robot arms (closed-loop kinematic chains). For example, see [122, 170].

6.0.3 Problem Statement

We now give a presentation of the multi-arm manipulation planning problem using a configuration space formalization. We consider only a single movable object, but for the rest, our presentation is general.

The environment is a 3D workspace $\mathcal{W}$ with p robot arms $\mathcal{A}_i$ $(i = 1, \cdots, p)$, a single movable object $\mathcal{M}$, and q static obstacles $\mathcal{B}_j$ $(j = 1, \cdots, q)$. The object $\mathcal{M}$ can only move by having one or several robots grasp and carry it to some destination.

Let $\mathcal{C}_i$ and $\mathcal{C}_{obj}$ be the C-spaces (configuration spaces) of the arms $\mathcal{A}_i$ and the object $\mathcal{M}$, respectively [144, 130]. Each $\mathcal{C}_i$ has dimension n_i, where n_i is the number of degrees of freedom of the robot $\mathcal{A}_i$, and $\mathcal{C}_{obj}$ is 6D. The *composite C-space* of the whole system is $\mathcal{C} = \mathcal{C}_1 \times \cdots \times \mathcal{C}_p \times \mathcal{C}_{obj}$. A configuration in $\mathcal{C}$, called a *system configuration*, is of the form $(\boldsymbol{q}_1, \cdots, \boldsymbol{q}_p, \boldsymbol{q}_{obj})$, with $\boldsymbol{q}_i \in \mathcal{C}_i$ and $\boldsymbol{q}_{obj} \in \mathcal{C}_{obj}$. In the example of Fig. 1, $p = 3$ and $n_i = 6$, for all i.

We define the *C-obstacle region* $\mathcal{CB} \subset \mathcal{C}$ as the set of all system configurations where two or more bodies in $\{\mathcal{A}_1, \cdots, \mathcal{A}_p, \mathcal{M}, \mathcal{B}_1, \cdots, \mathcal{B}_q\}$ intersect.[1] We describe all bodies as closed subsets of $\mathcal{W}$; hence, $\mathcal{CB}$ is a closed subset of $\mathcal{C}$. The open subset $\mathcal{C} \setminus \mathcal{CB}$ is denoted by $\mathcal{C}_{free}$ and its closure by $cl(\mathcal{C}_{free})$.

For the most part we require that the arms, object, and obstacles do not contact one another. However, $\mathcal{M}$ may touch stationary arms and obstacles for the purpose of achieving static stability. $\mathcal{M}$ may also touch arms when it is being moved, in order to achieve grasp stability; then $\mathcal{M}$ can only make contact with the last link of each grasping arm (grasping may involve one or several arms). No other contacts are allowed.

This leads us to define two subsets of $cl(\mathcal{C}_{free})$:
- The *stable space* $\mathcal{C}_{stable}$ is the set of all legal configurations in $cl(\mathcal{C}_{free})$ where $\mathcal{M}$ is statically stable. $\mathcal{M}$'s stability may be achieved by contacts between $\mathcal{M}$ and the arms and/or the obstacles.
- The *grasp space* $\mathcal{C}_{grasp}$ is the set of all legal configurations in $cl(\mathcal{C}_{free})$ where one or several arms rigidly grasp $\mathcal{M}$ in such a way that they have sufficient torque to move $\mathcal{M}$. $\mathcal{C}_{grasp} \subset \mathcal{C}_{stable}$.

[1] We regard joint limits in $\mathcal{A}_i$ as obstacles that only interfere with the arms' motions.

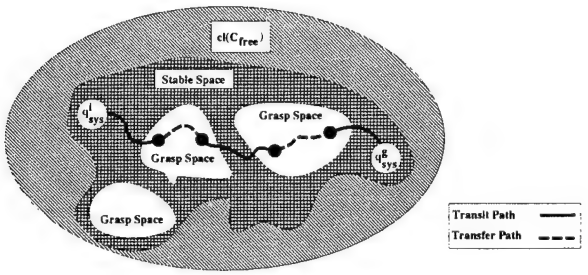


Figure 2. Components of a manipulation path and their relation to the subspaces of $cl(\mathcal{C}_{free})$.

There are two types of paths, transit and transfer paths, which are of interest in multi-arm manipulation:
- A *transit path* defines an arms' motion that does not move $\mathcal{M}$. Along such a path $\mathcal{M}$'s static stability must be achieved by contacts with obstacles and/or stationary arms. Examples of such a path involve moving an arm to a configuration where it can grasp $\mathcal{M}$ or moving an arm to change its grasp of $\mathcal{M}$. A transit path lies in the cross-section of $\mathcal{C}_{stable}$ defined by the current fixed configuration of $\mathcal{M}$.
- A *transfer path* defines an arms' motion that moves $\mathcal{M}$. It lies in the cross-section of $\mathcal{C}_{grasp}$ defined by the attachment of $\mathcal{M}$ to the last links of the grasping arms. During a transfer path, not all moving arms need grasp $\mathcal{M}$; some arms may be moving to allow the grasping arms to move without collision.

A *manipulation path* is an alternate sequence of transit and transfer paths that connects an initial system configuration, $\mathbf{q}^i_{sys}$, to a goal system configuration, $\mathbf{q}^g_{sys}$ (see Fig. 2). Some paths in this sequence may be executed concurrently as long as it does not yield collisions.

In a multi-arm manipulation planning problem, the geometry of the arms, movable object, and obstacles is given, along with the location of the obstacles. The goal is to compute a manipulation path between two input system configurations.

6.0.4 Planning Approach

We now describe our approach for solving the multi-arm manipulation planning problem. This approach embeds several simplifications, so that the corresponding planner is not fully general. Throughout our presentation, we carefully state the simplifications that we make. Some of them illustrate the deep intricacies of multi-arm manipulation planning.

Overview: A manipulation path alternates transit and transfer paths. Each path may be seen as the plan for a subtask of the total manipulation task. This yields the following two-stage planning approach: first, generate a series of subtasks to achieve the system goal configuration; second, plan a transit or transfer path for each subtask. An informal example of a series of subtasks is: grab $\mathcal{M}$, carry it to an intermediate location, change grasp, carry $\mathcal{M}$ to its goal location, ungrasp.

Unfortunately, identifying a series of subtasks that can later be completed into legal paths is a difficult problem. How can one determine whether a subtask can be completed without actually completing it? We settle for a compromise. Rather than planning for suitable transit *and* transfer subtasks, we focus solely on identifying a sequence of transfer tasks that are *guaranteed* to be completed into transfer paths. In fact, in the process of identifying these tasks, the planner also generates the corresponding transfer paths. With the transfer tasks specified, the transit tasks are immediately defined: they link the transfer paths together along with the initial and goal system configuration. Consequently, it only remains to compute the corresponding transit paths.

The assumption underlying this approach is that there exists a legal transit path for every transit task (since they are chosen without any guarantee that a transit path exists for them). Fortunately, in a 3D workspace, this is often the case. If the assumption is not verified, the planner may try to generate another series of transfer tasks, but in our current implementation it simply returns failure.

Restrictions on grasps: To simplifiy the selection of transfer tasks, we impose two main restrictions on grasps:

1. We consider two types of movable objects. An object of the first type can be moved using a single arm, for any grasp of the object. An object of the second type requires being grasped by two arms to move. The type of the movable object $\mathcal{M}$ is given as input to the planner.

2. The various possible grasps of $\mathcal{M}$ are given as a finite *grasp set*. Each grasp in this set describes a rigid attachment of the last link of an arm with $\mathcal{M}$ (first type of movable object), or a pair of such attachments (second type of movable object).

Generating transfer tasks: The generation of the transfer tasks is done by planning a path τ_{obj} of $\mathcal{M}$ from its initial to its goal configuration. During the computation of τ_{obj}, all the possible ways of grasping $\mathcal{M}$ are enumerated and the configurations of $\mathcal{M}$ requiring a regrasp are identified.

The planner computes the path τ_{obj} so that $\mathcal{M}$ avoids collision with the static obstacles $\mathcal{B}_j$. This is done using RPP (Randomized Path Planner), which is thus a component of our planner. RPP is described in detail in [68, 130].

RPP generates τ_{obj} as a list of adjacent configurations in a fine grid placed over $\mathcal{C}_{obj}$ (the 6D C-space of $\mathcal{M}$), by inserting one configuration after the other starting with the initial configuration of $\mathcal{M}$. The original RPP only checks that each inserted configuration is collision-free. To ensure that there exists a sequence of transfer paths moving $\mathcal{M}$ along τ_{obj}, we have modified RPP. The modified RPP also verifies that at each inserted configuration, $\mathcal{M}$ can be grasped using a grasp from the input grasp set. This is done in the following way. A *grasp assignment* at some configuration of $\mathcal{M}$ is a pair associating an element of the grasp set defined for $\mathcal{M}$ and the identity of the grasping robot(s). Note that the same element of the grasp set may yield different grasp assignments involving different robots. The planner enumerates all the grasp assignments at the initial configuration of $\mathcal{M}$ and keeps a list of those which can be achieved without collision between the grasping arm(s) and the obstacles, and between the two grasping arms (if $\mathcal{M}$ must be moved by two arms). We momentarily

ignore the possibility that the grasping arm(s) may collide with the other arms. The list of possible grasp assignments is associated to the initial configuration. Prior to inserting any new configuration in the path being generated, RPP prunes the list of grasp assignments attached to the previous configuration by removing all those which are no longer possible at the new configuration. The remaining sublist, if not empty, is associated with this configuration and appended to the current path.

If during a down motion of RPP (a motion along the negated gradient of the potential field used by RPP) the list of grasp assignments pruned as above vanishes at all the successors of the current configuration (call it $\boldsymbol{q}_{obj}$), the modified RPP resets the list attached to $\boldsymbol{q}_{obj}$ to contain all the possible grasp assignments at $\boldsymbol{q}_{obj}$ (as we proceed from the initial configuration). During a random motion (a motion intended to escape a local minimum of the potential), the list of grasp assignments is pruned but is constrained to never vanish. In the process of constructing τ_{obj}, the modified RPP may reset the grasp assignment list several times.

If successful, the outcome of RPP is a path τ_{obj} described as a series of configurations of $\mathcal{M}$, each anotated with a grasp assignment list. The path τ_{obj} is thus partitioned into a series of subpaths, each connecting two successive configurations. It defines as many transfer tasks as there are distinct grasp assignments associated with it. By construction, *for each such transfer task, there exists a transfer path satisfying the corresponding grasp assignment.* The number of regrasps along the generated path τ_{obj} is minimal, but RPP does not guarantee that this is the best path in that respect.

Details and comments: The condition that the same grasp assignment be possible at two neighboring configurations of $\mathcal{M}$ does not guarantee that the displacement of $\mathcal{M}$ can be done by a short (hence, collision-free) motion of the grasping arm(s). An additional test is needed when the set of grasps between two consecutive configurations is pruned. In our implementation, we assume that each arm is a non-redundant 6-DOF arm. Hence, an arm can attain a grasp with a small number of different postures, which can easily be computed using the arm's inverse kinematics. We include the posture of each involved arm in the description of a grasp assignment. Hence the same combination of arms achieving the same grasp, but with two different postures of at least one arm, defines two distinct grasp assignments. Then a configuration of $\mathcal{M}$, along with a grasp assignment, uniquely defines the configurations of the grasping arms. The resolution of the grid placed across $\mathcal{C}_{obj}$ is set fine enough to guarantee that the motion between any two neighboring configurations of $\mathcal{M}$ results in a maximal arm displacement smaller than some prespecified threshold.

A transfer path could be obstructed by the arms not currently grasping $\mathcal{M}$. Dealing with these arms can be particularly complicated. In our current implementation, we assume that each arm has a relatively non-obstructive configuration given in the problem definition (in the system shown in Fig. 1, the given non-obstructive configuration of each arm is when it stands vertical). Prior to a transfer path, all arms not involved in grasping $\mathcal{M}$ are moved to their non-obstructive configurations. The planner nevertheless checks that no collision occurs with them during the construction of τ_{obj}.

Perhaps the most bl:" ant limitation of our approach is that it does not plan for regrasps at configurations of $\mathcal{M}$ where it makes contact with obstacles (as we said, τ_{obj} is computed free of collisions with obstacles). Since the object cannot levitate, we require that $\mathcal{M}$ be held at all times during regrasp. We assume that if $\mathcal{M}$ requires more than one robot to move, any subset of a grasp is sufficient to achieve static stability during the regrasp. For example, if a grasp requires two robots, anyone of these robots, alone, achieves static stability, allowing the other robot to move along a transit path. An obvious example where this limitation may prevent our planner from finding a path is when the robot system contains a single arm; no regrasp is then possible.

RPP is only probabilistically complete [68]. If a path exists for $\mathcal{M}$, it will find it, but the computation time cannot be bounded in advance. Furthermore, if no path exists, RPP may run for ever. Nevertheless, for a rigid object (as is the case for $\mathcal{M}$), RPP is usually very quick to return a path, when one exists. Hence, we can easily set a time limit beyond which it is safe to assume that no path exists. Other path planners could possibly be used in place of RPP.

Generation of transit paths: The transfer tasks identified as above can be organized into successive layers, as illustrated in Fig. 3. Each layer contains all the transfer tasks generated for the same subpath of τ_{obj}; the transfer tasks differ by the grasp assignment. Selecting one such task in every layer yields a series of transit tasks: the first consists of achieving the first grasp from the initial system configuration; it is followed by a possibly empty series of transit tasks to change grasps between two consecutive transfer tasks; the last transit task is to achieve the goal system configuration. Hence, it remains to identify a grasp assignment in each layer of the graph shown in Fig. 3, such that there exist transit paths accomplishing the corresponding transit tasks.

Assume without loss of generality that all arms are initially at their non-obstructive configurations. Our planner first chooses a transfer task (anyone) in the first layer. Consider the transit task of going from the initial system configuration to the configuration where the arms achieve the grasp assignment specified in the chosen transfer task, with $\mathcal{M}$ being at its initial configuration. The coordinated path of the arms is generated using RPP. If this fails, a new attempt is made with another transfer task in the first layer; otherwise, a transfer task is selected in the second layer. The connection of the system configuration at the end of the first transfer task to the system configuration at the start of this second transfer task forms a new transit task.

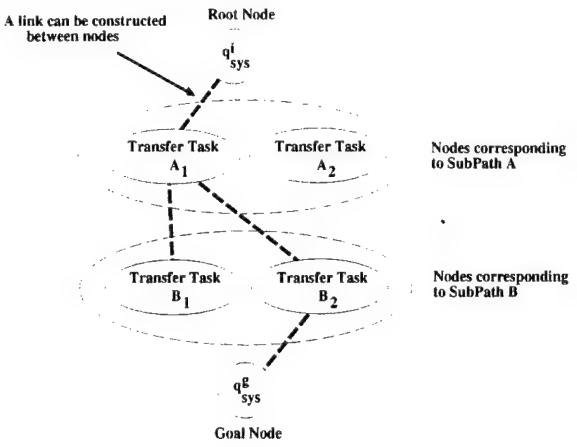
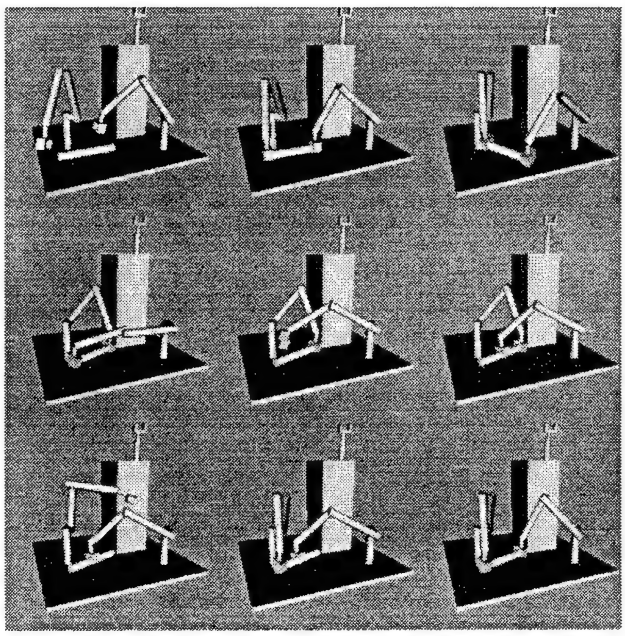


Figure 3. The directed and layered graph.

The transit task between two transfer tasks is more difficult to solve. To understand the difficulty, imagine the case where $\mathcal{M}$ is a long bar requiring two arms to move. Assume that the robot system contains only two arms and that the bar can be grasped at its two ends and at its center. Consider the situation where the bar is grasped at its two ends and the regrasp requires swapping the two arms. This regrasp is not possible without introducing an intermediate grasp. For example: arm 1 will ungrasp one end of the bar and regrasp it at its center (during this regrasp, arm 2 will be holding the bar without moving); then arm 2 will ungrasp the bar and regrasp it at the other end; finally, arm 1 will ungrasp the center of the bar and will regrasp it at its free end. Fig. 4 illustrates this example.



(d) (e) (f)

(g) (h) (i)

Figure 4. An example illustrating the complexity of changing grasps.

We address this difficulty by breaking the transit task between two transfer tasks into smaller transit subtasks. Each transit subtask consists of going from one grasp assignment to another in such a way that no two arms use the same grasp at the same time. In this process, we allow arms not involved in the first and last assignment to be used. We start with the first grasp assignment and we generate all the potential grasp assignments that we may be able to achieve from it (assuming the corresponding transit paths exist). We generate the successors of these new assignments, and so on until we reach the desired assignment (the one used in the next transfer task). For each sequence that achieves this desired assignment, we test that it is actually feasible by using RPP to generate a transit path between every two successive grasp assignments. We stop as soon as we obtain a feasible sequence. The concatenation of the corresponding sequence of transit paths forms the transit path connecting the two considered transfer tasks. We then proceed to link to the next layer of transfer tasks.

When we reach a transfer path in the last layer, its connection to the goal system configuration is carried out in the same way as the connection of the initial system configuration to the first layer.

6.0.5 Examples of Generated Paths

We implemented the above approach in a planner written in C and running of a DEC Alpha workstation under UNIX. All experiments so far were conducted with a robot system made of 3 identical arms, each with 6 revolute joints. The non-obstructive configuration of each arm is one where the arm stands vertical.

Fig. 1 shows a manipulation path generated by the planner for an L-shaped object. This object requires two arms to move it. The object is taken through the window in the large obstacle located in the middle of the workspace. Notice that in the change of grasp, at least one arm is holding the object at all times. For this path, it took 45 seconds to identify the transfer tasks and 30 additional seconds to complete the manipulation path. For the generation of τ_{obj}, the object's C-space was discretized into a $100 \times 100 \times 100$ grid. For the generation of the transit paths, the joint angles of the arms were discretized into intervals of 0.05 radians. The grasp set contains 64 grasps, yielding grasp assignment lists up to around 15,000 elements.

Fig. 4 shows a manipulation path generated for a long bar requiring two arms to move it. This example illustrates the complexity of changing grasps. For this path, it took 25 seconds to identify the transfer tasks and an additional 20 seconds to complete the manipulation path. The same discretizations as above were used. The grasp set of the long bar contains 24 grasps, yielding grasp assignment lists up to around 3,000 elements.

Fig. 5 shows a manipulation path found for a T-shaped object. This object requires a single arm to move it, and only two arms are used along the computed path. One arm first grasps the object at one end of the T. It passes the object to another arm that grasps it at its other end and brings it to

its goal configuration. For this path, the planner took 40 seconds to identify the transfer tasks and then another 25 seconds to complete the manipulation path. The same discretizations as above were used. The grasp set of the T-shaped object contains 49 grasps. It yields grasp assignment lists up to around 2,000 elements.

For these examples, in computing the transit paths RPP uses the sum of the angular joint distances to the goal configuration as the guiding potential. However, in computing τ_{obj} RPP uses an NF2-based potential with three control points [130]. For both cases of finding the transit paths and τ_{obj}, we limit the amount of computation spent in RPP to three backtrack operations [130], after which the planner returns failure. Failure to find τ_{obj} results in the immediate failure to find a manipulation path. Similarly, a failure to find transit paths to link together the layers of transfer paths results in a failure to find a manipulation path. The time for the planner to report its failure depends on the problem, with some examples being from 30 seconds to a few minutes.

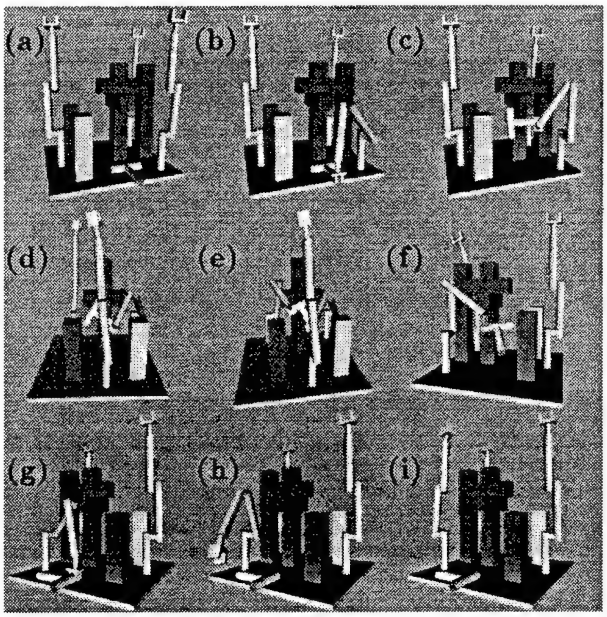


Figure 5. Another example of a manipulation path. The object is T-shaped and requires only one arm to manipulate it.

6.0.6 Conclusion

Multi-arm manipulation planning is a new motion planning problem with application in various tasks, such as assembly and construction. We have presented an approach for solving this complicated problem. Our approach embeds several simplifications yielding an implemented planner that is not fully general. However, experiments with this planner show that it is quite reliable and fast in finding manipulation paths, when such paths exist, making it suitable as an interactive tool to

facilitate robot programming. We believe the robust nature of the planner is the result of careful consideration of the general manipulation problem, the introduction of reasonable simplifications, and the appropriate utilization of the efficient randomized path planner.

Most of our experiments so far have been conducted with simulated mechanical arms. However, we have completed a first integration of the planner with a software module simulating human arms which incorporates the inverse kinematics of such arms [129]. The result is a system that computes natural human-arm manipulation motions. The goal of this work is to help in the interactive generation of animated video sequences for studying task ergonomy. Also, such videos would usefully replace cumbersome instruction manuals accompanying some assembly kits.

Our next step is to connect our planner to the controller of our 3-PUMA system and experiment with the paths generated by the planner to verify our assumptions in the real world. Later, we hope to include the ability to regrasp the movable object by having the arms place it in a stable configuration against some obstacles. We also plan to use existing results to automatically compute the grasp set of an object from its geometric model. The technique described in [125] to deal with multiple movable objects in a 2D space should also be applicable in our planner. Furthermore, like in [125], we plan to allow parallel execution of consecutive transfer and transit paths as long as this parallelism does not yield collisions. Finally, we hope to incorporate considerations of torque constraints and dynamics issues in the planning process. Indeed, for the regrasping actions we currently assume that a single arm is sufficient to hold the object statically, when actually there may be certain configuations where the actuator torques are too limited. Furthermore, allowing the object to slip in the grasp of the robots due to gravitational or inertial effects may be another way of changing grasps.

Chapter 7

Motion Planning with Uncertainty: A Landmark Approach

This chapter is based on the paper "Motion Planning with Uncertainty: A Landmark Approach," by A. Lazanas and J.C. Latombe, published in Artificial Intelligence J., *76(1-2), 1995, 287-317.*

7.1 Introduction

Motion planning with uncertainty is a critical problem in robotics. Indeed, even the most complex models of the physical world cannot be perfectly accurate. Furthermore, increasing model complexity often adversely affects the ability of a robot to plan its actions efficiently. Thus, the use of simplified models seems to be the only practical way to proceed. All details omitted from these models unite to form uncertainty. A robot planner must produce plans that reliably achieve their goals despite errors, i.e., differences between the models and the real world.

Taking uncertainty into account at planning time is essential when potential errors are comparable to or larger than the tolerances allowed by the task. This is the case, for example, for mechanical part-mating tasks, where both errors and tolerances are usually small, and mobile robot navigation tasks, where both errors and tolerances may be large. For such tasks, classical path planning methods [130], which use simple geometric models while assuming null uncertainty, are clearly insufficient. At best, they produce paths that require frequent replanning to deal with discrepancies detected by sensors during execution. But, due to errors in sensing, they may also lead the robot to incorrectly believe that it has achieved some expected state or, on the contrary, that it has failed to achieve this state. To generate reliable plans, the planner must choose courses of actions whose execution is guaranteed to make enough knowledge available to allow the robot to correctly identify the states it traverses, despite execution-time errors in control and sensing [73]. The overall task

of the planner is to select an adequate set of states, associate appropriate motion commands with these states, and synthesize state-recognition functions. All three subtasks are interdependent.

One often expects too much from a system reasoning in the presence of uncertainty. This attitude has often led to engineering brittle complicated systems whose actual capabilities are difficult to assess. Experiments may show that these systems work beautifully on some tasks, but fail miserably on some others, without us being able to predict one outcome or the other. We believe that the design of a reliable system dealing with uncertainty must be based on bounded, well-defined expectations of what may actually happen in the real world, allowing some desirable operational properties to be achieved. In this chapter we propose a precisely defined motion planning problem with uncertainty and a sound, complete, and polynomial algorithm that solves this problem. By sound, we mean that the planner only generates correct plans, which are guaranteed to succeed if some predefined assumptions bounding uncertainty are satisfied. By complete, we mean that the planner returns a correct plan whenever one exists; otherwise, it declares failure. By polynomial, we mean that the worst-case asymptotic time complexity of the planner is a polynomial function of the size of the input problem.

While several motion planners with uncertainty have been proposed (e.g., [143, 177, 76, 164, 96, 133, 91, 115, 145, 97, 92, 132, 176]), soundness and completeness are provable for only a few of them. Furthermore, most known sound and complete planning algorithms take exponential time in some measure of the size of the input problem (typically, the complexity of the robot's environment) [80, 93], which makes them virtually inapplicable. In fact, it has been shown that these motion planners attack problems which are inherently intractable [78]. In our planner, like in previous planners, soundness and completeness is proven under the assumption that all errors are restricted to lie within bounded uncertainty intervals. We achieve polynomiality both by making assumptions in the problem formulation that eliminate the source of intractability and by using algorithms that take advantage of these assumptions. The key notion underlying these assumptions is that of a landmark, an "island of perfection" in the robot configuration space where position sensing and motion control are accurate. This notion considerably simplifies the selection of the set of states that the robot may traverse and the synthesis of the state-recognition functions. It mainly reduces planning to selecting motion commands to navigate from landmark to landmark until the goal is achieved. For a given problem, our planner generates a plan represented in the form of a collection of reaction rules. Each rule is attached to some region of the robot configuration space and specifies the motion command to be executed if this region is attained. Such a plan is reminiscent of a reaction plan as proposed in [84, 95, 171].

Creating landmarks requires some prior engineering of the robot and/or its environment. Though we believe that such engineering is unavoidable to build reliable practical robot systems, it is important that its cost be reduced as much as possible (after all, industrial mobile robots tracking wires have been in use for many years). This leads us to analyze how uncertainty bounds affect plans. We show that the structure of a correct plan must change only at a finite number of critical values of these bounds. Hence, any additional engineering reducing uncertainty bounds without crossing a critical value is useless. We also show that some uncertainty in control and sensing can be allowed in landmark regions without affecting the correctness of a generated plan.

Although the concepts underlying our planner are more general, the current geometric techniques it uses are restricted to two-dimensional configuration spaces. This limits the application of the planner, mainly to circular mobile robots on flat terrain. The planner was implemented for such robots. We have successfully experimented with it on many problems, both in simulation and using a real robot in our laboratory environment. Since the plans generated by our planner are provably correct, the main goal of our experiments with a real robot was to verify that our concept of a landmark did not induce inacceptable engineering costs. Actually, we will see that our work led us to design different sorts of relatively low-cost landmarks.

The main contribution of this chapter is to provide solid theoretically and experimentally proven foundations for implementing mobile robot navigation systems, along with specific algorithmic planning techniques. We believe, however, that a potential impact of our work is to redefine the purpose of experimentation in robot planning with uncertainty (more generally, in autonomous systems). When the assumptions underlying the design of a planner are left implicit or informal, experimentation is aimed at verifying that the planner works adequately on a representative sample of tasks. When assumptions are precisely stated and the planning algorithm is provably sound and complete for these assumptions and reasonably efficient, as is the case of our planner, experimentation is aimed at verifying that satisfying the assumptions does not entail excessive engineering. This second purpose results in an easier experimental task. It may also lead to defining standard sets of assumptions, known as inducing acceptable engineering costs, hence allowing subsequent research to be specifically aimed at developing more efficient planners operating under such assumptions.

Section 2 situates our work relative to previous related work. Section 3 precisely states the planning problem addressed in this chapter, lists the main results obtained, and illustrates them with an example. Section 4 outlines the principle of our planning algorithm. Section 5 instantiates this algorithm in the case where the uncertainty in control is fixed. Sections 6 and 7 deal with variable control uncertainty and present algorithms to generate one-step and multi-step plans, respectively. Section 8 analyzes assumptions made in our work and discusses its relevance to actual robot navigation. Section 9 describes the implementation of our planning techniques and their experimentation with a real mobile robot. Section 10 suggests other extensions of the planner.

7.2 Background

Motion planning with uncertainty has been a research topic in robotics for almost two decades. Two main application domains have been considered: part mating for mechanical assembly and mobile robot navigation. Various approaches have been proposed, which are applicable to one domain, or the other, or both. They include skeleton refinement [143, 177, 76, 164], inductive learning from experiments [96], iterative removal of contacts [133, 91, 115], and preimage backchaining [145, 153, 97]. Skeleton refinement, inductive learning, and iterative removal of contacts essentially operate in two phases: a motion plan is first generated assuming no uncertainty and then transformed to deal with uncertainty. Instead, preimage backchaining takes uncertainty into account throughout the whole planning process. In principle, it can tackle problems where uncertainty shapes the structure

of a plan to the extent that it cannot be generated by transforming an initial one produced under the no-uncertainty assumption, which is often the case for mobile robot problems. Our work is a direct continuation of a series of work on preimage backchaining. We focus most of our discussion on this series.

Preimage backchaining considers the following class of motion planning problems [145]: A plan is a sequence (more generally, an algorithm) of motion commands, each defined by a *commanded direction of motion* d and a *termination condition* **TC**. When the robot executes such a command in free space, it moves along a direction contained at each instant in a cone of half-angle θ about d and stops as soon as **TC** becomes true. (In contact with an obstacle, the robot may stop or slide, depending on the particular control law that is used. In this chapter we will simply forbid contacts with obstacles.) The angle θ is the largest expected control error and models the *directional uncertainty* of the robot. The termination condition **TC** is a boolean function of sensory data s. At any one time, these data measure physical parameters (e.g., the robot position coordinates) with some error. The actual parameter vector lies anywhere in a region $\mathcal{U}(s)$ modeling the robot's *sensory uncertainty.* A planning problem is specified by a workspace model, an *initial region* where the robot is known to be prior to executing the plan, a *goal region* in which the robot must stop when plan execution terminates, the directional uncertainty θ, and the sensing uncertainty $\mathcal{U}$. A *correct* plan is one whose execution guarantees the robot to stop in the goal region despite errors in control and sensing. A *sound* planner is one which generates only correct plans. A *complete* planner is one which returns a correct plan whenever one exists, and failure otherwise.

The above problem formulation admits many variants. For example, one may introduce time and consider uncertainty in the robot velocity, allowing the construction of more sophisticated termination conditions [153, 97]. The workspace model (e.g., the location and the shape of the obstacles) may also be subject to errors, yielding a third type of uncertainty [92]. For the sake of simplicity we will not discuss these variants here (see [130]).

The *preimage* of a goal region for a given motion command $\mathbf{M} = (d,\mathbf{TC})$ is the set of all points in the robot's configuration space such that if the robot starts executing the command from any one of these points, it is guaranteed to reach the goal and stop in it. Preimage backchaining consists of constructing a sequence of motion commands $\mathbf{M}_i$, $i = 1,\ldots,n$, such that, if $\mathcal{P}_n$ is the preimage of the goal for $\mathbf{M}_n$, $\mathcal{P}_{n-1}$ the preimage of $\mathcal{P}_n$ for $\mathbf{M}_{n-1}$, and so on, then $\mathcal{P}_1$ contains the initial region.

One source of difficulty in computing preimages is the interaction between goal reachability and goal recognizability. The robot must both reach the goal (despite directional uncertainty) and stop in the goal (despite sensing uncertainty). Goal recognition, hence the termination condition of a command, often depends on the region from where the command is executed. This region, which is precisely the preimage of the goal for that command, also depends on the termination condition. This recursive dependence was noted in [145]. Despite this difficulty, Canny [80] described a complete planner with very few restrictive assumptions in it. This planner generates an r-step plan by reducing the input problem to deciding the satisfiability of a semi-algebraic formula with $2r$ alternating existential and universal quantifiers. Such a decision takes double exponential time in r. Moreover, the smallest r for which a plan may exist grows with the complexity of the environment.

Actually, various forms of the above motion planning problem have been proven intrinsically hard [78, 155, 80].

At the expense of completeness, Erdmann [97] suggested that goal reachability and recognizability be treated separately by identifying a subset of the goal, called a *kernel*, such that when this subset is attained, goal achievement can be recognized (by **TC**) independently of the way it has been achieved. He defined the backprojection of a region $\mathcal{R}$ for a motion command **M** as the set of all points such that, if the robot executes **M** starting at any one of these points, it is guaranteed to reach $\mathcal{R}$. He proposed an $O(n \log n)$ algorithm to compute backprojections in the plane when the obstacles are polygons bounded by n edges. An implemented planner based on this approach is described in [132].

Once the kernel of a goal has been identified, a remaining issue is the selection of the commanded direction of motion to attain this kernel, since the backprojection of the kernel depends on this direction. The planner described in [132] only considers a finite number of regularly spaced directions; hence, it is incomplete, and usually not very efficient. The continuous set of backprojections of a region $\mathcal{R}$ for all possible directions of motion is called the *nondirectional backprojection* of $\mathcal{R}$. In the plane (i.e., for a point robot moving in the plane), Donald [93] showed that, as far as planning is concerned, this set is sufficiently described by a polynomial number of (directional) backprojections. These backprojections are computed at critical directions of motion where the topology of the backprojection's boundary changes. Donald proposed an $O(n^4 \log n)$ algorithm, with n being the number of obstacle edges, to compute nondirectional backprojections and embedded this algorithm into a polynomial one-step planner. Briggs [75] reduced the time complexity of computing a nondirectional backprojection to $O(n^2 \log n)$. Fox and Hutchinson [103] extended the algorithm to exploit visual constraints and allow visual compliant motions.

However, even when nondirectional backprojections are used, one last difficulty to construct a multi-step planner is backchaining. The difficulty comes from the fact that backchaining introduces a twofold variation: when the commanded direction of motion varies, both the backprojection of the current kernel and the kernel of this backprojection (which will be used at the next backchaining iteration) vary. In this chapter (as in [137, 138]) we deal with this difficulty by introducing *landmarks*. We define a landmark region as a subset of the robot's configuration space where position sensing and motion control are perfect, while outside landmark regions sensing is null and control is imperfect with errors bounded by some given directional uncertainty θ (see Section 3). We show that backchaining is then reduced to iteratively computing a relatively small set of (directional) backprojections for a growing set of landmark regions. This result directly yields a complete planning algorithm that takes polynomial time in the complexity of the landmark and obstacle regions. Previously, Friedman [104] had proposed another polynomial multi-step planner for a compliant point robot in a polygonal workspace by assuming that sensing exactly detects when the robot traverses line segments joining vertices of the workspace.

The above definition of uncertainty corresponds to treating control and sensing errors as random variables with uniform distribution over bounded domains. The advantage of bounding errors over, say, a Gaussian distribution model is that it yields the neat and convenient notion of a correct

motion plan. However, a given planning problem may admit no such plans, while it could have admitted one if uncertainty had been set slightly smaller. Furthermore, when correct plans exist, they may be overly complex. To deal with this drawback, Donald [92] proposed the notion of an Error Detection and Recovery Strategy, defined as an r-step plan ($r > 1$) that attains the goal whenever it is recognizably reachable and signals failure whenever chances that it serendipitously achieve the goal vanish. Erdmann [99] introduced randomized plans whose execution converges toward the goal with probability one. In this chapter we propose a different approach which consists of allowing the directional uncertainty θ to vary over some interval. We propose computational tools to build planners that can select values of θ according to various optimization criteria. For example, θ may be actually tunable by the robot. (Most robots can achieve such control; they may reduce θ by slowing down and devoting more computation time to determine which commands to send to the actuators, allowing more accurate dynamic models of the robot's mechanical structure to be used.) Then, if no correct plans exist for some value of θ, or if the existing plans are too complex, the planner may try to use smaller values of θ. However, reducing directional uncertainty generates some cost (e.g., lower speed), which the planner should strive to minimize. As we will see, allowing θ to vary has other interesting applications, such as dealing with unexpected obstacles and generating probabilistic plans. As indicated earlier, understanding the dependency of correct plans on directional uncertainty may also facilitate the task of engineering the robot and its environment.

Our notion of a landmark corresponds approximately to a recognizable feature of the workspace that induces a field of influence (the landmark region); if the robot is in this field, it senses the landmark. Similar notions have been previously introduced in the literature with different names, e.g., landmarks [142], atomic regions [77], signature neighborhoods [152], perceptual equivalent classes [94], sensory uncertainty field [176], and visual constraints [114, 103]. Our landmarks are mainly aimed at simplifying the selection of the set of states that the robot may traverse and the synthesis of the state-recognition functions. Instead, research like the one described in [94] is aimed at automating state selection and state recognition. Although it has great potential in reducing the engineering cost and the limitations associated with our landmarks, it is not clear as yet whether it can yield time-efficient planners.

Over the past decade, there has also been a substantial amount of work at reducing uncertainty (mainly position uncertainty) while a robot is moving. For example, for mobile robots, many techniques have been proposed to combine the estimates provided by both dead-reckoning and environmental sensing (e.g., see [66, 90, 141, 181]). These techniques address the problem of tracking a selected motion plan as well as possible, not the problem of generating this plan. The goal of planning in the presence of uncertainty is to make sure that executing the plan will reveal enough information to guarantee reliable execution. Notice also that planning and execution may use different models. For instance, modelling errors as random variables uniformly distributed over compact subsets makes sense at planning time. However, it may be preferable to use more sophisticated probabilistic distributions at execution time to better use all sources of information then available.

Reaction plans have been previously proposed as a way to deal with uncertainty at planning time [171, 84, 95]. Such a plan consists of goal-oriented rules triggered by the data available at

execution time. In particular, Schoppers [171] developed the notion of a universal plan whose rules cover all possible situations that may occur at execution time. The plans generated by our planner are very similar. They consist of motion commands attached to regions of the robot's configuration space. When such a region is entered by the robot, the associated motion command is executed. The interesting point is that our planner takes polynomial time, while the generation of universal plans is often believed to be exponential. However, unlike the planners mentioned above, our planner addresses a restricted family of planning problems in which uncertainty is well-bounded. Our planner achieves polynomiality by identifying criticalities allowing the originally continuous search space to be partitioned into a polynomial-size discrete search space.

7.3 Statement of Problem and Results

Problem Statement: The planning problem considered in this chapter is the following:

The robot is a point[1] moving in a plane, called the *workspace*, containing forbidden circular regions, the *obstacles*, and other circular regions, called the *landmark disks*. Both the obstacles and the landmark disks are stationary. The robot is not allowed to collide with any of the obstacles. The number of landmark disks is finite and equal to ℓ. The number of obstacle disks is also finite and in $O(\ell)$. (Throughout this chapter the number ℓ is used to measure the size of the input problem.)

The robot has perfect position sensing in the landmark disks and no sensing outside the landmark disks. It can move in either the *perfect* or the *imperfect* control mode. In the perfect mode, it navigates without error. In the imperfect mode, its actual direction of motion at any instant differs from the commanded direction of motion by an angle bounded by θ (directional uncertainty). The perfect mode is feasible only in landmark disks. If several such disks form a connected area, called a *landmark area*, the robot can move in the perfect mode over all this area. The imperfect mode is applicable everywhere. The robot has no sense of time (thus, the modulus of its velocity is irrelevant to the planning problem).

The value of θ is controllable by the robot in a connected interval $[\theta_{min}, \theta_{max}] \subset [0, \pi/2)$. A motion command in the imperfect control mode is specified as a triplet $(d, \theta, \mathcal{L})$, where $d \in S^1$ (the unit circle) is the commanded direction of motion, $\theta \in [\theta_{min}, \theta_{max}]$ is the directional uncertainty, and $\mathcal{L}$ a set of landmark disks defining the termination condition (the robot stops as soon as it enters one of these disks). $\mathcal{L}$ is called the *termination set* of the command. A decreasing function defines the cost of navigating with uncertainty θ. (Throughout this chapter, we will remain intentionally vague about the cost function. Our main goal is to show that controllable directional uncertainty can be handled by a planner.)

Prior to execution, the robot is known to lie anywhere in an initial region $\mathcal{I}$ that consists of one or several disks. The number of disks in $\mathcal{I}$ is small enough to be considered in $O(1)$. The robot must move into a given goal region $\mathcal{G}$, which may be any subset, connected or not, of the workspace,

[1] For a circular mobile robot, this is obtained by shrinking the robot to its center, while isotropically growing the obstacles by the robot's radius.

whose intersection with the landmark disks takes $O(\ell)$ time to compute.

The problem is to generate a sequence of motion commands to make the robot move from its initial position into the goal region and stop there. In doing so we wish to minimize the cost paid for the various choices of directional uncertainty in the imperfect motion commands.

Main Results: In the context of the above problem, the main results presented in this chapter are the following:

(i) We show that the four-dimensional set of backprojections of any given collection of landmark disks, for all directions of motion d in S^1 and all values of the directional uncertainty θ in $[\theta_{min}, \theta_{max}]$, is sufficiently represented (as far as planning is concerned) by a polynomial number of backprojections, each computed for a specific value of d and θ.

(ii) From result (i), we derive three polynomial planning algorithms:

- The first algorithm assumes given constant directional uncertainty. It is sound and complete, and produces plans minimizing the number of motion commands to be executed.

- The second algorithm assumes controllable directional uncertainty. It generates correct one-step motion plans maximizing directional uncertainty. For one-step plans, it is sound and complete.

- The third planner extends the second one and uses a greedy algorithm to generate multi-step plans in which each step allows maximal directional uncertainty. It is sound and complete.

(iii) All three algorithms have been implemented and we show sample runs obtained with the planners.

(iv) We briefly discuss other applications of the techniques described in this chapter: least commitment planning to deal with unexpected obstacles, planning with anisotropic uncertainty, and generation of probabilistic plans.

Examples: Figures 7.1 and 7.2 illustrate the problem formulation given above. Each figure depicts a plan generated by one of the implemented planners.

Example 1: Fig. 7.1 shows an example run with the first planner. The workspace contains 23 landmark disks (shown white or grey) forming 19 landmark areas, and 25 obstacle disks. The directional uncertainty is fixed and set to 0.09 radian. The initial (resp., goal) region is a small disk designated by $\mathcal{I}$ (resp., $\mathcal{G}$). The white disks are those with which the planner has associated imperfect motion commands to attain another landmark disk. The arrow attached to the initial region or a white disk is the commanded direction of such a command. There is at least one arrow per landmark area not intersecting the goal. The arrow attached to a landmark disk originates at a point called an *exit point.*

Hence, the plan consists of motion commands distributed over landmark disks. Its execution

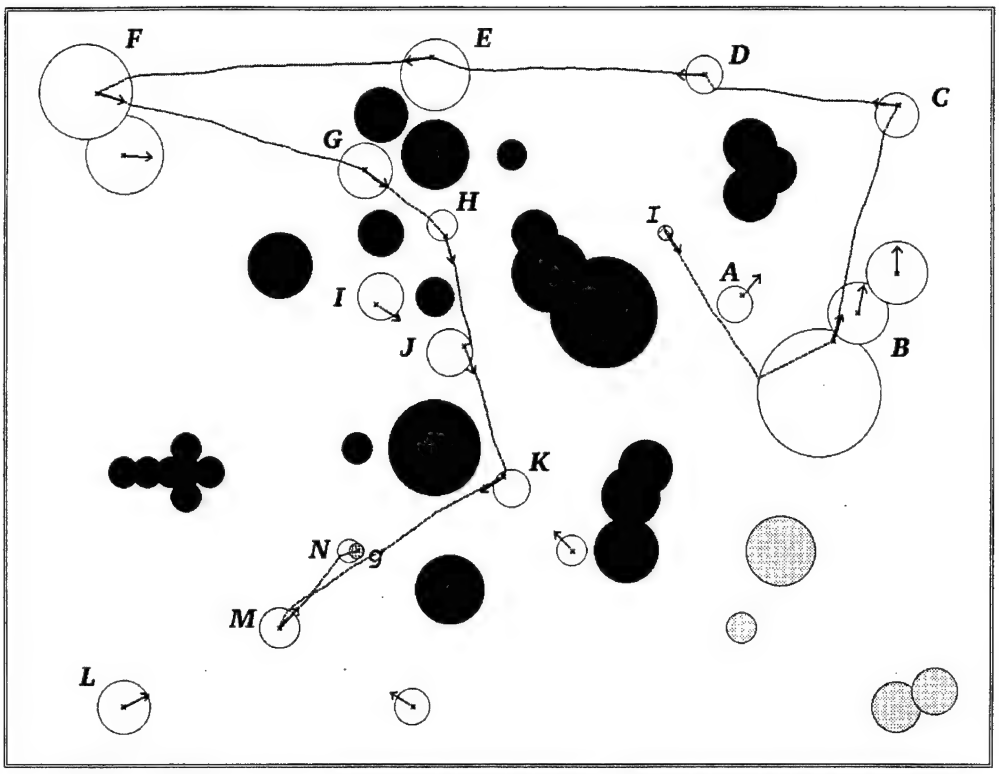


Figure 7.1: Planning Example 1

begins with performing the command attached to $\mathcal{I}$. When the robot reaches a landmark disk in the termination set of this command, it executes a perfect motion command, either to a point in the goal $\mathcal{G}$, if the landmark disk belongs to a landmark area intersecting $\mathcal{G}$, or to an exit point of the landmark area. In the second case, the imperfect motion command associated with that point is executed, and so on.

The figure shows the path produced by a sample execution of the plan (in simulation). This path first takes the robot from the initial region to the landmark area designated by B. From there, it successively attains and traverses the landmark areas marked C, D, E, F, G, H, J, K, M, and N. The number of imperfect motion commands executed along this path is 11. However, the generated plan could have necessitated up to 12 commands. Indeed, the termination set of the command from K contains L, M, and N. Another execution (with different control errors, but the same value of θ) could have caused the robot to reach L rather than M. The command attached to L would then have allowed the robot to reach M. No correct plans require less than 12 imperfect motion commands to be executed in the worst case.

Example 2: Fig. 7.2 shows a simple example run with the second planner. The workspace contains 5 landmark disks, with two landmark areas intersecting the goal $\mathcal{G}$, and 2 obstacle disks. The initial

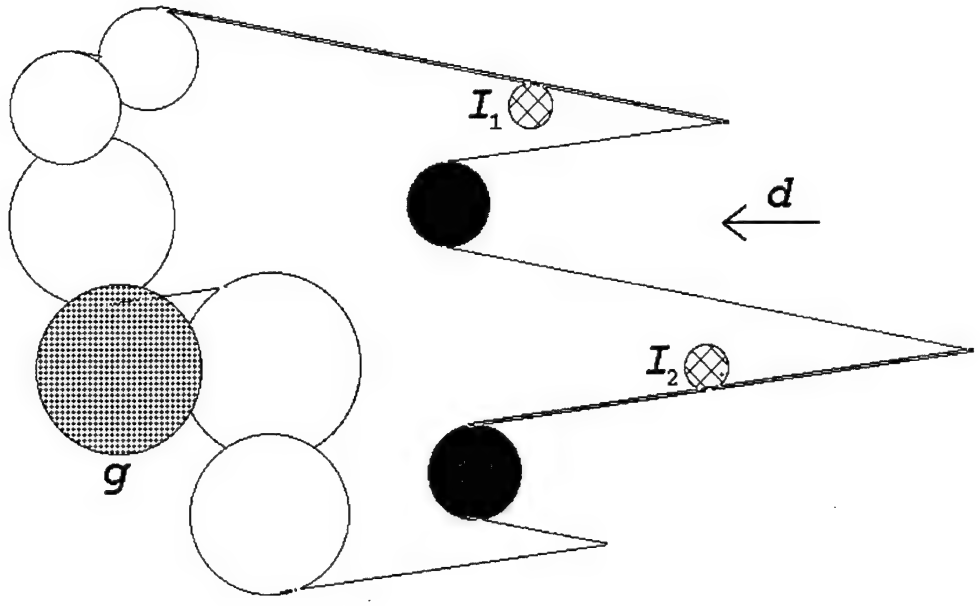


Figure 7.2: Planning Example 2

region consists of two disks, $\mathcal{I}_1$ and $\mathcal{I}_2$. The directional uncertainty is controllable in the interval [0.1,0.5] radian. The planner produces a one-step plan, i.e., a plan containing a single imperfect motion command $(d, \theta, \mathcal{L})$, with d shown in the figure, $\theta = 0.16$ radian, and $\mathcal{L}$ made up by the two landmark areas intersecting $\mathcal{G}$. The set of points in the workspace from which this command is guaranteed to reach $\mathcal{L}$ (i.e., the backprojection of $\mathcal{L}$) is outlined in the figure. It fully contains the initial-region disks $\mathcal{I}_1$ and $\mathcal{I}_2$. The value of the control uncertainty selected by the planner is maximum over all correct one-step plans. Any slight variation of the direction of motion and/or directional uncertainty would cause an initial-region disk to be partially outside the backprojection of $\mathcal{L}$.

Discussion: The above problem statement is certainly simplifcation of a real mobile robot navigation problem, but we think it is not oversimplified. We will discuss the assumptions made at greater length later in this chapter (mainly in Section 9). Before entering the technical details, let survey the main points of our future discussion:

- It is straightfoward to generalize our algorithms in order to deal with generalized polygonal landmark and obstacle regions, that is, regions bounded by straight and circular edges.

- Assumptions made outside landmark regions concerning sensing and control are somewhat conservative. However, they do not prevent more sophisticated models to be used at execution time. Hence, our planner may be somewhat pessimistic. But it can be run for several uncertainty bounds,

including optimistic ones.

- Assumptions made inside landmark regions are anti-conservative, since neither sensing or control can ever be perfect. However, we will see that some uncertainty is acceptable within these regions, yielding the concept of a "generalized landmark."

7.4 Outline of a Planning Algorithm

Under the assumptions made in the problem of the previous section, if a connected set of landmark disks, i.e., a landmark area, intersects the goal region, the robot can move into the goal from any point in this set, in the perfect control mode. We call the union of the landmark areas intersecting the goal, the *extension* of the goal;[2] the other landmark disks are called the *intermediate-goal disks.* If the goal does not intersect any landmark disk, then it is considered unachievable, since the robot cannot sense its achievement. By definition of the goal extension, the robot cannot move into it using the perfect control mode.

Given a goal $\mathcal{G}$, we first compute its extension $\mathcal{E}(\mathcal{G})$. If the initial region $\mathcal{I}$ lies entirely in $\mathcal{E}(\mathcal{G})$, no further planning effort is needed since a correct plan to achieve the goal has already been found. Indeed, in a landmark area, a plan is simply a geometric path, whose computation is straightforward. Such a plan is called a *zero-step plan.* (In the following, we will measure the length of a plan by the number of imperfect motion commands it contains.)

The backprojection of $\mathcal{E}(\mathcal{G})$ for the pair (d, θ) is the maximal set of points such that executing the imperfect motion command $(d, \theta, \mathcal{E}(\mathcal{G}))$ from any of these points is guaranteed to reach $\mathcal{E}(\mathcal{G})$. If there does not exist a zero-step plan to achieve $\mathcal{G}$, the planner may try to find a pair (d, θ) such that the initial region $\mathcal{I}$ is contained in the backprojection of $\mathcal{E}(\mathcal{G})$ for (d, θ). If one such pair (d, θ) is found, the command $(d, \theta, \mathcal{E}(\mathcal{G}))$ starting from anywhere within $\mathcal{I}$ is guaranteed to attain and terminate in $\mathcal{E}(\mathcal{G})$. From there a zero-step plan will achieve the goal $\mathcal{G}$. We call this plan a *one-step* plan.

Fig. 7.3 shows an example with five landmark disks (displayed grey and white), and two obstacle disks (black). The white disks form the extension of the goal $\mathcal{G}$; the grey disks are intermediate-goal disks. The initial region $\mathcal{I}$ consists of a single disk. This disk is totally included in the backprojection outlined for direction d and uncertainty θ. The backprojection is bounded by circular arcs and straight edges. The latter are supported by rays erected from landmark and obstacle disks. These rays are tangent to the disks and parallel to the directions $d \pm \theta$. Two intersecting rays form a *spike.* The backprojection of Fig. 7.3 has two spikes. (See [138] for a more detailed description of a backprojection.) The boundary of the backprojection consists of $O(\ell)$ arcs and edges.

A one-step plan may not exist, or may not be desirable if its cost is too high. Then the planner can attempt to create a *multi-step* plan iteratively. At each iteration, it selects a pair (d, θ),

[2] In previous papers [137, 138], we called this set the kernel of the goal. However, this terminology was somewhat confusing, since the word 'kernel' had previously been used in preimage backchaining, with a different meaning.

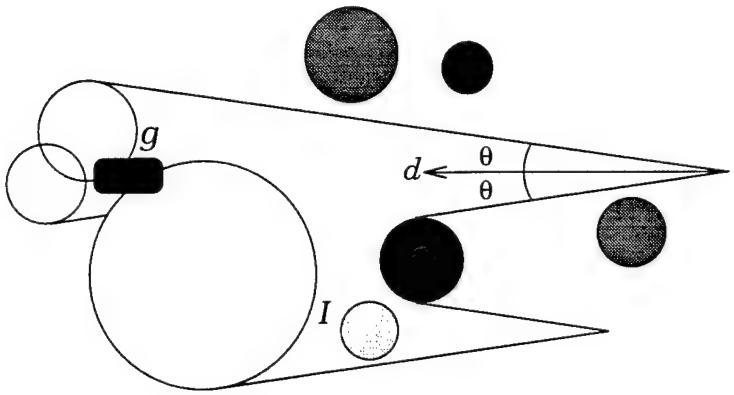


Figure 7.3: Backprojection of a goal extension

such that the corresponding backprojection of the current goal extension intersects one or more intermediate-goal disks (landmark disks not contained in the current goal extension). All the landmark disks in the landmark areas containing these intersected disks are added to the goal extension to generate a larger extension for the next iteration; they are no longer intermediate-goal disks. The backprojection of the new goal extension is computed, and so on, until the initial region is contained in a backprojection, or no new intermediate-goal disks can be intersected, in which case a correct plan cannot possibly exist.

Let $s \in O(\ell)$ be the number of landmark areas; the number of iterations performed by the planner is bounded by s. The computation of every backprojection can be done in time $O(\ell \log \ell)$ using a traditional sweep-line algorithm given in [138]. Determining which initial-position disks are contained in the backprojection and which intermediate-goal disks are intersected by it is done while the backprojection is being computed, within the same asymptotic time complexity.

The construction of the search space explored by the above algorithm requires discretizing $S^1 \times [\theta_{min}, \theta_{max}]$, i.e., the continuous $d\theta$-space. In other words, we must answer the following question: At each iteration, which values of (d, θ) should the planner consider? In the next section we show that the $d\theta$-space can be decomposed into a finite number of cells and that one pair (d, θ) need be considered in each cell to ensure that the planner is complete. From this result we derive a finite search space that can be explored exhaustively, if necessary.

Since we allow θ to vary, one may wonder if the planner can ever return failure. The answer is obviously yes if the lower bound on θ is strictly positive. It remains yes, even if θ is allowed to become zero and $\mathcal{G}$ intersects a landmark disk. For example, this happens if the workspace contains a single landmark disk intersecting $\mathcal{G}$, and $\mathcal{I}$ consists of a single disk that is bigger than the landmark disk.

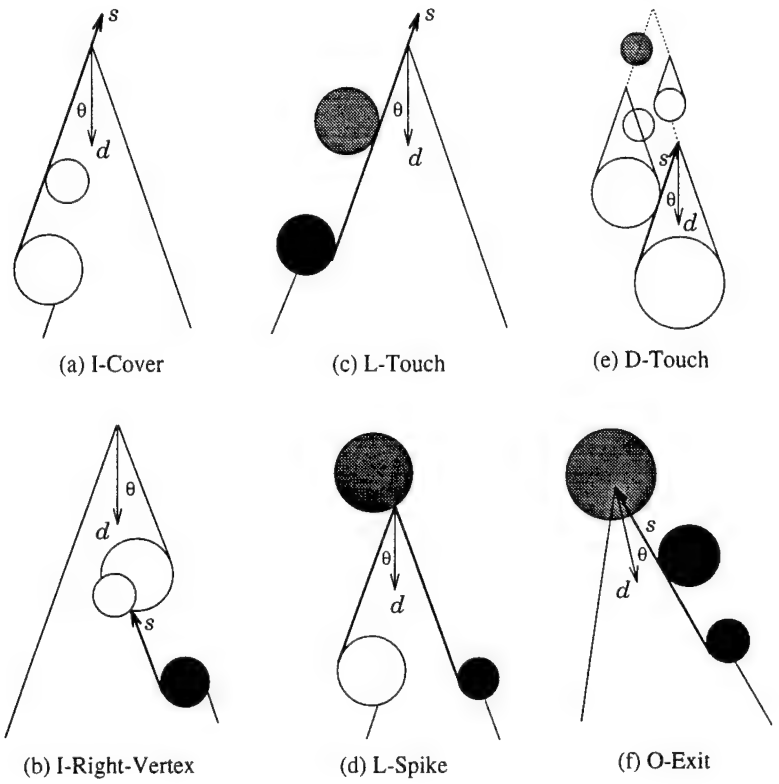


Figure 7.4: Representative critical events

7.5 Building the Discrete Search Space

Each planning iteration requires selecting a pair (d, θ) such that the backprojection of the current goal extension either contains the initial region $\mathcal{I}$, or intersects intermediate-goal disks. We now show that the $d\theta$-space can be partitioned into an arrangement of *cells* of dimensions 2, 1, and 0 (points), which are regular in the following sense: The backprojection of the goal extension for any pair (d, θ) in a cell C contains the same initial-position disks and intersects the same intermediate-goal disks as the backprojection for any other pair (d', θ') in C. The number of cells is polynomial in the number of landmark and obstacle disks.

Let us assume that no landmark disk intersects an obstacle disk (all disks are considered to be closed subsets). The arrangement of cells is created by a network of curves corresponding to the following *critical events* (some of them are illustrated in Fig. 7.4):

- *I-Cover event*: A left ray of the backprojection is tangent to an initial-region disk, with this disk on its right-hand side.

- *I-Leave event*: A right ray of the backprojection is tangent to an initial-region disk, with this disk on its left.

- *I-Left-Vertex event*: The endpoint of a left ray of the backprojection coincides with the entry intersection point[3] of an initial position disk by a disk contained in the goal extension.

- *I-Right-Vertex event*: The endpoint of a right ray of the backprojection coincides with the exit intersection point of an initial position disk by a disk of the goal extension.

- *L-Touch event*: A left ray of the backprojection is tangent to an intermediate-goal disk, with this disk on its left.

- *L-Exit event*: A right ray of the backprojection is tangent to an intermediate-goal disk, with this disk on its right.

- *L-Spike event*: A spike of the backprojection lies on the boundary of a intermediate-goal disk.

- *D-Touch event*: A left ray of the backprojection is tangent to a disk of the goal extension, with this disk on its left.

- *D-Exit event*: A right ray of the backprojection is tangent to a disk of the goal extension, with this disk on its right.

- *O-Touch event*: A left ray of the backprojection is tangent to an obstacle disk, with this disk on its left.

- *O-Exit event*: A right ray of the backprojection is tangent to an obstacle disk, with this disk on its right.

Consider an I-Cover event. Let s be the slope of the left ray tangent to the initial-position disk. The equation of the curve defined by this event is:

$$\theta = d - s + \pi \pmod{2\pi}.$$

Similar linear equations (with slopes equal to ± 1) can be established for the curves defined by the I-Leave, I-Left-Vertex, I-Right-Vertex, L-Touch, and L-Exit, D-Touch, D-Exit, O-Touch, and O-Exit events. The equation of the curve of an L-Spike event is significantly more complicated. However, we can show that it is of the form:

$$\theta = f_{spike}(d),$$

where f_{spike} has a single maximum and at most one intersection with any line of slope ± 1 (see [139] for a proof of these claims). We call f_{spike} a *spike curve.*

The number of I-Cover and I-Leave curves is $O(\ell)$, the number of I-Left-Vertex, I-Right-Vertex, L-Touch, L-Exit, D-Touch, D-Exit, O-Touch and O-Exit curves is $O(\ell^2)$, and the number of L-Spike curves is $O(\ell^3)$. These curves determine an arrangement of $O(\ell^6)$ cells. Fig. 7.5 shows a cell arrangement with $\theta \in (0, \pi/2)$.

[3]When a disk δ_1 intersects another one, δ_2, we define the *entry intersection point* of δ_2 by δ_1, as the point where we enter δ_2 when we move counterclockwisely along the boundary of δ_1. The point where we exit δ_2 is the *exit intersection point.*

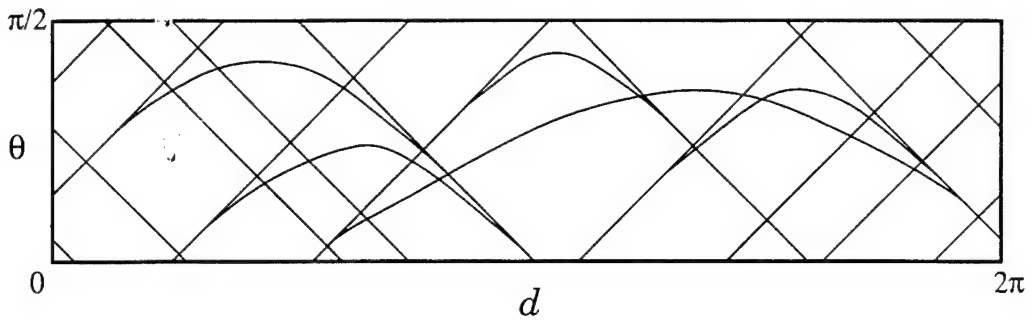


Figure 7.5: Cell arrangement in the $d\theta$-plane.

Allowing landmark disks to intersect with obstacle disks would simply require considering additional critical events. It would not change the asymptotic complexity of the cell arrangement, nor the complexity of the planners presented in the following sections. For the sake of simplicity, we will not describe this simple extension here.

7.6 Planning with Constant Directional Uncertainty

Let us first consider the case where the directional uncertainty is fixed to some given value θ_f. We previously considered this case in [138].

In the $d\theta$-plane the line $\theta = \theta_f$ intersects the critical curves at $O(\ell^3)$ points (critical commanded directions of motion), which partition the set S^1 of values of d into $O(\ell^3)$ cells of dimensions 1 and 0 (open intervals and points). The planning algorithm sketched in Section 7.4 can proceed iteratively in a kind of breadth-first manner, as follows:

At each iteration, decompose S^1 into cells, select an arbitrary commanded direction of motion in each cell, and compute the corresponding backprojection (along with the initial-region disks that this backprojection contains and the intermediate-goal disks that it intersects). For every computed backprojection B, for every intermediate-goal disk L that this backprojection intersects, select an exit point in the intersection $B \cap L$. Associate the imperfect motion command $(d, \theta_f, \mathcal{E})$ with this point, where d is the direction used to compute the backprojection B and $\mathcal{E}$ is the current goal extension. (This command is part of the plan being generated. During plan execution, if the robot ever reaches the landmark area containing the disk L, it will move to the exit point of L in the perfect control mode; it will then execute the imperfect motion command $(d, \theta_f, \mathcal{E})$ associated with this point.) If any of the computed backprojections fully contains the initial region, return success. Otherwise, include all the landmark areas containing one or several intermediate-goal disks intersected by a backprojection in the current goal extension to form the extension for the next iteration.

The computation of a backprojection takes time $O(\ell \log \ell)$. Since there are $O(\ell^3)$ cells, the time

complexity of a planning iteration is $O(\ell^4 \log \ell)$. Up to s iterations are required, where $s \in O(\ell)$ is the number of landmark areas. Hence, the algorithm returns a plan, if one exists, in time $O(s\ell^4 \log \ell)$; otherwise, it declares failure in the same time.

After the kth planning iteration ($k = 1, 2, ...$), the goal extension is the largest set of landmark disks from which $\mathcal{G}$ can be attained reliably by executing at most k imperfect motion commands. Hence, after at most s iterations, the goal extension eventually contains all landmark disks from which the goal can be reliably attained by executing a finite number of motion commands. At any iteration, if a backprojection of the current goal extension contains $\mathcal{I}$, the planner will find it. Hence the planner is complete.

The outcome of the planner is a "distributed plan" made of imperfect motion commands attached to exit points in landmark areas (see Fig. 7.1). The algorithm may generate several exit points in the same landmark area or disk (if this area or disk is intersected by several backprojections computed at the same iteration). In principle, the planner could keep a single exit point per landmark area, and discard the others. However, it may be preferable to keep several exit points among which the robot can choose at execution time in order to minimize the length of the paths performed in the perfect control mode in the landmark areas [138].

Several executions of the same plan may lead the robot to perform different sequences of commands, since control errors (in the interval $[0, \theta_f]$) may yield the same command to terminate in one landmark area or another (in the termination set of the command). The number of imperfect motion commands that the robot may have to execute is upper bounded by the number of iterations performed by the planner. The definition of a backprojection and the construction of the cells guarantee that this number is minimal over all possible correct plans for the given initial and goal regions. Furthermore, after the execution of any sequence of steps, the subset of the motion plan that may still be used to attain the goal has the same property. In this sense, the plans produced by the above algorithm are optimal.

The above algorithm proceeds in a kind of breadth-first fashion by computing one backprojection per cell at every iteration. One can easily imagine variants based on other search strategies. However, not all strategies yield optimal plans.

The time complexity of the planner, $O(s\ell^4 \log \ell)$, is one order of magnitude greater than the complexity of a similar planner described in [138]. The difference comes from the fact that the above planner recomputes a backprojection from scratch in every cell of the decomposition of S^1. Instead, the planner of [138] computes a first backprojection and incrementally modifies this backprojection as it scans the cells in S^1. However, this requires tracking all the changes in the backprojection topology, yielding 33 different types of events that include the 11 types listed in Section 7.5. There are significant practical advantages in having less events to consider. In particular, the above algorithm is simpler to implement than the one of [138]. Because it recomputes backprojections from scratch in every cell, it is also less sensitive to floating-point computation errors.

7.7 One-Step Planning with Controlled Uncertainty

Now let the directional uncertainty θ be controllable by the robot within the given interval $[\theta_{min}, \theta_{max}]$. We first address the one-step plan generation problem. Multi-step planning will be considered in the next section.

To generate a one-step plan the planner only needs to find a pair (d, θ), with $\theta \in [\theta_{min}, \theta_{max}]$, such that the backprojection of the goal extension $\mathcal{E}(\mathcal{G})$ for the imperfect motion command $(d, \theta, \mathcal{E}(\mathcal{G}))$ fully contains the initial region $\mathcal{I}$. Thus, the planner can discard the L-event critical curves in the $d\theta$-plane. It can also discard the O-event critical curves, because these are only used in conjunction with L-Spike event curves.[4] The remaining I- and L-event curves (all straight lines) define an arrangement of $O(\ell^4)$ cells. In every cell, the planner can select an arbitrary pair (d, θ) and compute the corresponding backprojection. In the worst case, the planner scans all the cells. Hence, it returns a plan or declares failure in time $O(\ell^5 \log \ell)$. The resulting planner is complete.

In general, if a one-step plan exists, it is preferable to generate one which allows maximal control uncertainty. The value of θ for such a plan can only be θ_{max} or the θ-coordinate of the intersection of two critical lines. This yields the following planning algorithm: Set θ to θ_{max} and compute the backprojection of $\mathcal{E}(\mathcal{G})$ for each cell of the arrangement intersecting the line $\theta = \theta_{max}$. If one backprojection contains $\mathcal{I}$, return success (and the corresponding value of d). If no backprojection contains $\mathcal{I}$, scan the intersections of I- and L-event lines verifying $\theta \in [\theta_{min}, \theta_{max}]$ in decreasing order of their θ-coordinates. For every intersection point (d, θ), compute the backprojection of $\mathcal{E}(\mathcal{G})$. If this backprojection contains $\mathcal{I}$ return success, otherwise consider the next intersection point. Return failure if all intersections have been considered without success.

Using a sweep-line technique to scan the intersection points, this algorithm takes output-sensitive time $O((\ell^2 + c\ell) \log \ell)$, where $c \in O(\ell^4)$ is the rank (in decreasing order) of the value of θ selected among the θ-coordinates of the $O(\ell^4)$ intersection points of the event lines.

Fig. 7.2 shows a motion command generated by the above algorithm.

7.8 Multi-Step Planning with Controlled Uncertainty

To generate a multi-step plan, we can use the following greedy algorithm:

At every iteration, sweep a line parallel to the d-axis in order to find the highest value of θ such that there exists a backprojection of the current goal extension which either contains the initial region or intersects *one* intermediate-goal disk. In the second case, add the disks in the landmark area intersected by the backprojection to the goal extension and introduce the corresponding new event curves into the arrangement. Repeat this procedure until the computed backprojection contains all initial-position disks, or no more intermediate-goal disks can be intersected. In the latter case return failure.

[4] O-events serve only to indicate the limits of existence of a spike. For more details see [139].

At every iteration, the algorithm need not compute the intersections of the L-Spike event curves among themselves, since these intersection points cannot give rise to the highest value of θ we are looking for. (Remember we only seek the *first* intermediate-goal disk to be intersected.) Furthermore, the algorithm must consider intersection points among event curves at most once over all iterations (see proof in [139]). Hence, the total complexity of the planner is $O(s\ell^5 \log \ell)$. The output plan has s steps at most.

Fig. 7.6 illustrates the operation of this algorithm with an example. The workspace contains 7 landmark disks **A** through **G**, and 4 obstacles. The goal $\mathcal{G}$ intersects with landmark disk **A**. The initial region $\mathcal{I}$ consists of a single disk. The directional uncertainty lies in the interval $[0.0, 0.5]$ radian. This means that we are not interested in uncertainty angles above 0.5 radian because the robot cannot do worse than that. Using $\theta = 0.5$, the planner first finds a correct motion command to reach **A** from the landmark disks **B** and **C**. The extension of goal then consists of **A**, **B**, and **C**. For the commanded direction shown in the upper-left figure and $\theta = 0.41$, the backprojection of this goal extension touches **D**, which is added in turn to the goal extension. **E** is then touched by a backprojection for $\theta = 0.28$ (upper-right figure). **F** is touched by a backprojection for $\theta = 0.32$ (lower-left figure). Both disks **F** and **G** are then added to the goal extension, since they intersect. At this point, all landmark disks are in the goal extension. The initial-position disk is then covered by a backprojection for $\theta = 0.1$ (lower-right figure).

At every iteration, the algorithm maximizes directional uncertainty to achieve the current extension of the goal. In general, the generated plan, if any, would not minimize a given cost function. Generating minimum-cost plans seems intrinsically harder. For example, it may require the selection of a smaller directional uncertainty at an early iteration, if this choice allows larger values of the uncertainty to be selected at subsequent iterations.

7.9 Relevance to Robot Navigation

At this point it is worth looking again at the assumptions made in the problem statement of Section 7.3. As mentioned earlier, this statement is certainly a simplification of a real mobile-robot navigation problem, but we think it is not oversimplified. Indeed, it captures the essence of the actual problem and, as argued below, the assumptions made are more realistic than it may appear at first glance.

Some assumptions do not correspond to actual limitations of the approach. For example, the restriction of landmark and obstacle regions to unions of disks could easily be removed to allow these areas to be described as generalized polygons (regions bounded by simple closed curves made of straight and circular edges). This extension only requires some straightforward adaptation in the construction of the critical curves. Adding critical events also allows landmark and obstacle regions to touch each other. The fact that the robot is a point in a two-dimensional space is slightly more serious since it limits the configuration space of the actual robot to be two-dimensional. Hence, either the actual robot is circular, or it can only translate. The underlying principles of our approach

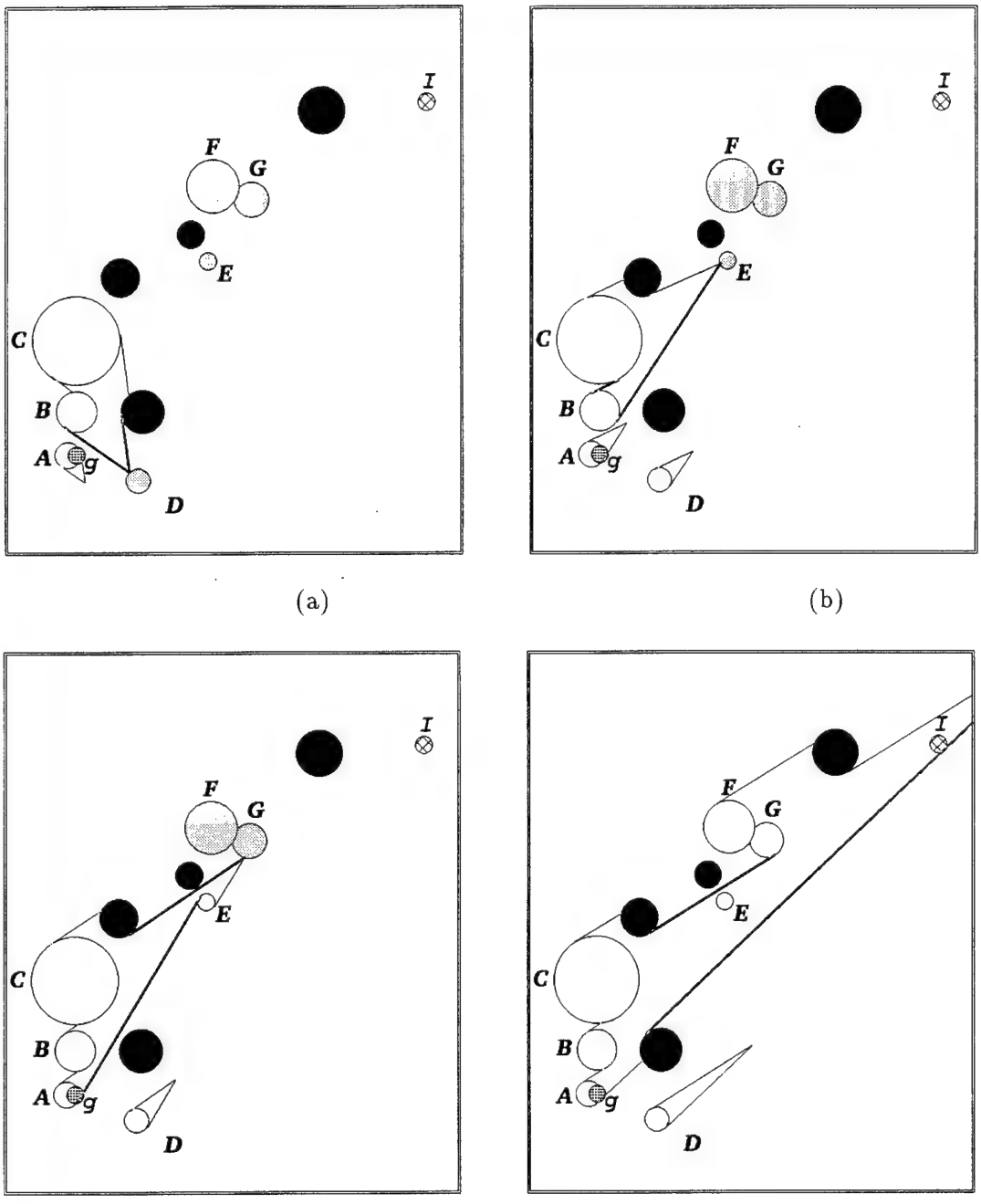


Figure 7.6: Multi-Step Greedy Planning

remain valid in higher dimensional spaces, but critical curves become critical hyper-surfaces. They are more difficult to define and lead to more complex arrangements that are more complex to compute.

Perhaps the most disturbing assumptions are those used in the definition of the landmark areas, namely that control and position sensing errors are null within these areas, while position sensing is inexistent outside. We now focus our discussion on these assumptions in the context of a mobile robot.

A typical mobile robot uses two techniques to continuously estimate its position: dead-reckoning and environmental sensing. Environmental sensing provides pertinent information only when some characteristic features of the workspace (i.e., "landmarks") are visible by the sensors. Then the robot knows its position with good accuracy. When no or few features are visible, the robot mostly relies on dead-reckoning, which yields cumulative errors that we model by the directional uncertainty cone. Our assumption that sensing outside landmark areas is null is usually conservative, but it does not prevent the robot's navigation system from using all available sensing information at execution time to better determine the robot's current position. (As mentioned in Section 7.2, the navigation system does not have to use the same model as the planner; it may use a more sophisticated one, if this is possible.) In the worst case, the no-sensing assumption outside landmark areas may only lead our planners to return failure, while reliable plans exist in practice and, possibly, could have been found by more powerful planners able to deal with more sophisticated models.

On the contrary, the assumption that control is perfect in the landmark areas is anti-conservative; we believe, however, that it is a reasonable one, provided that we choose safe features and equip the robot with the right sensors. Landmark areas with sharp boundaries can be obtained by introducing artificial landmarks (e.g., radio or magnetic beacons) and/or thresholding an estimate of the sensing uncertainty. For example, the notion of a "sensory uncertainty field" (SUF) is introduced in [176]. At every possible point q in the configuration space, the SUF estimates the range of possible errors in the sensed configuration that the navigation system would compute by matching the sensory data against a prior model of the workspace, if the robot was at q. The SUF is computed at planning time from a model of the robot's sensing system. Thresholding it yields landmark areas. Uncertainty in the location of a landmark and/or fuzziness of its boundary can be handled by defining a smaller landmark area for our planner.

More interestingly, however, in our first planner (Section 7.6), perfect control and sensing in landmark areas are not strictly needed. Indeed, once the robot enters a landmark area it is sufficient that it reaches an "exit region" of non-zero measure prior to executing the next imperfect command. This region is the intersection of the backprojection that yielded this command with the landmark area. The exit point selected by the algorithm of Section 7.6 is just one particular point in this area. Position sensing uncertainty in a landmark area could be half the radius of the largest disk fully contained in the exit region of the landmark area without putting plan execution at risk. Thus, although the planner assumed perfect sensing in landmark areas, we can now create these areas by engineering the workspace in such a way that the sensors just provide the information

that is needed by the plan (see [100] for a similar idea).

Going further, the definition of a landmark can even be modified without having to significantly change the planning techniques developed above. For example, we could accept a landmark disk (or generalized polygon) L such that the robot knows at any time if it is inside or outside L, but if it is in L it does not know where. If during planning, a backprojection intersects L, this is not sufficient to include L in the extension of the goal. L must be completely contained in the backprojection. The critical events for which this has to be tested are exactly those used to check the containment of an initial-region disk.

The above variant of a landmark can be generalized into the notion of a *generalized landmark*, as follows: Consider a region L, such that if the robot is in L, it knows that it is in L and it has a way to accurately reach a point P in L. However, the robot may not have perfect control in L, nor perfect position sensing, except when it has reached P. For example, the landmark may be a wall; when the robot makes contact with the wall, its bumpers detect contact, but it still does not know precisely its position. By tracking the wall in some given direction, it can detect the end of the wall, a point where it knows its position with excellent accuracy. This point (or a small disk around this point) can be considered as a landmark area that the planner will try to enclose in a backprojection. If this happens, the whole region L will be added to the goal extension.

The above remarks hint a hierarchical way to organize landmarks and show that there exist multiple ways of engineering our planners in order to solve problems not covered by the definition of Section 7.3.

7.10 Implementation and Experimentation

Simulation: We implemented the planners described in Sections 7.6, 7.7, and 7.8 in a program written in the C language and running on a DEC-station 5000. The only major issue in this implementation concerns the computation of the maxima of the L-Spike event curves and their intersections with event lines. We have not been able to calculate analytical expressions for these points. Therefore, our planners use traditional numerical techniques, which requires some care to avoid inconsistent topological results in constructing backprojection. In order to visualize the plans generated by the planners, we have developed a simple simulator. Imperfect motion commands are discretized into short segments and a directional error is randomly selected for each segment. The simulator allows the user to tune segment length and error distribution in the interval $[0, \theta]$, to generate various sample runs.

The examples shown in Fig. 7.1, 7.2, and 7.6 were generated using the planners described in Section 7.6, 7.7, and 7.8, respectively. The respective computation times for these examples were 3.5 minutes,[5] 10 seconds, and 80 seconds. We have run the planner on many other examples. Though our evaluation of the theoretical complexity of the planner is rather high, we were able to

[5]Actually, this time was obtained with the planner described in [138] and mentioned in Section 7.6.

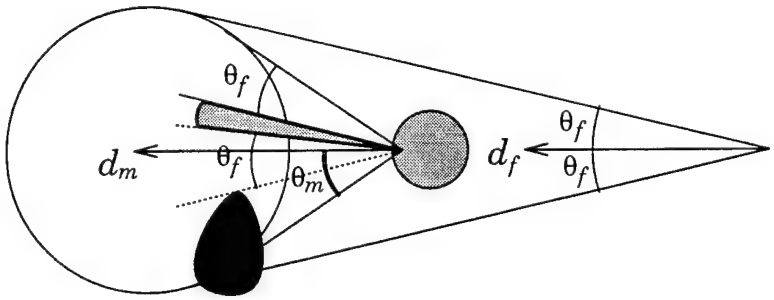


Figure 7.7: Least commitment navigation technique.

handle problems with up to 50 landmark disks reasonably fast (of the order of minutes).

In general, our experiments tend to show that the planners are more efficient than the asymptotic worst-case complexity analysis of the previous sections suggests. We believe that deeper combinatorial analysis of critical curves, cost amortization over iterations, and output-sensitive complexity evaluation should make it possible to produce tighter complexity results.

Real robot: We also have experimented with the first planner using a NOMAD 200 robot of Nomadic Technologies. A three-whell synchronous drive mechanical allows the robot to execute nonholonomic paths with zero turning radius. Hence, assuming no control errors, the robot can track polygonal paths by stopping and rotating at every vertex.

*** Description of corner landmarks. Example.

*** Other possible landmarks.

7.11 Other Extensions

In Section 9 we mentioned several important, but rather straightforward extensions of the current planners. Here, we briefly discuss other possible extensions.

Dealing with Unexpected Obstacles: Assume that the robot workspace may contain unexpected obstacles, i.e., obstacle regions that are not in the planner's model. Assume further that the robot sensors can detect the unexpected obstacles interfering with a motion command just before this command is executed. (By interfering obstacle, we mean one that intersects the set of all positions that the robot may traverse or reach during the execution of the command. This set is called the *forward projection* of the motion command [77, 80, 93]. A slight variation of the algorithm computing backprojections can be used to compute forward projections.)

Fig. 7.7 shows an example with a single goal-extension disk in white, an intermediate-goal disk in

grey, and an unexpected obstacle in black. If θ_f is the given constant directional uncertainty of the robot, the planner of Section 7.6 computes a backprojection $\mathcal{B}_f$ of the goal extension that intersects the intermediate-goal disk and provides the corresponding value of d, e.g., the direction d_f shown in the figure. However, if the robot executes a motion commanded along d_f starting from any point in the intersection of $\mathcal{B}_f$ and the intermediate-goal disk, it may hit the unexpected obstacle.

Instead, we can compute the maximal value of θ, θ_m, for which there exists a backprojection $\mathcal{B}_m$ of the goal extension that intersects the intermediate-goal disk, yielding a direction of motion d_m. Let $C(d_m, \theta_m)$ be the cone of half-angle θ_m about d_m. Let the robot move in the perfect control mode to the intersection of $\mathcal{B}_m$ with the intermediate-goal disk before executing an imperfect motion command of uncertainty θ_f toward the goal extension. In the absence of unexpected obstacles, the axis of any cone of half-angle θ_f contained in $C(d_m, \theta_m)$ is a commanded direction of motion guaranteed to achieve the goal extension with directional uncertainty θ_f. In the presence of unexpected obstacles, as is the case in Fig. 7.7, if there exists a cone of half-angle θ_f not intersecting the unexpected obstacles, its axis is a valid commanded direction.

Generating Non-Correct Plans: The planners of Sections 7.6 and 7.8 are complete for multi-step plans. If they fail to generate a plan, the input problem has no correct solution. However, before failing, they may associate imperfect motion commands to landmark areas from which the goal can reliably be achieved. In other words, these planners always return a plan. If the problem admits no solution, the plan is incomplete, i.e., no motion command is associated with the initial region. The robot may nevertheless attempt to reach a landmark area with an associated command by executing a random walk. In the plane, the probability that such a motion enters a set of disks converges toward 1 as time grows. The larger this set, the faster the convergence.

Another way to deal with failure is to allow uncertainty to decrease below its minimal value until a backprojection of the current goal extension intersects an intermediate-goal disk or contains the initial region. Any motion command planned in this way is no longer guaranteed to attain the goal extension it is aimed at. This approach thus yields the concept of a non-correct plan that maximizes probability of success.

One major drawback of a plan generated as above is that its execution may fail in a non-recognizable way. Indeed, a non-correct motion may miss all the disks in its termination set and continue for ever. Introducing some awareness of time is one way to address this drawback. Another approach, inspired by Donald's EDR strategies [92], is to make sure that every non-correct imperfect motion command inserted in the plan will either succeed or fail recognizably. Such commands have a termination set that they are guaranteed to reach, but some disks in this set are not part of the goal extension at the time they were selected. The generation of non-correct plans that recognize failure, while maximizing probability of success, is an interesting topic still requiring additional research.

Planning with Anisotropic Uncertainty: In some applications, the directional uncertainty θ depends on the commanded direction of motion. For example, this can happen when a wheeled

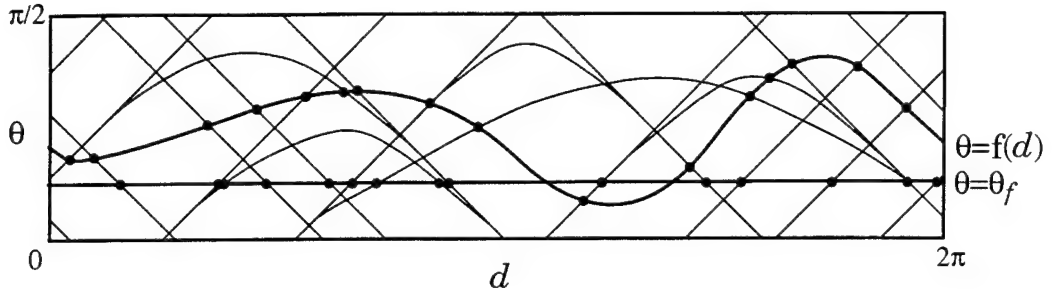


Figure 7.8: Modeling anisotropic directional uncertainty

robot moves on a carpet, or more generally in any flow field (e.g., a river). If we know the function $\theta = f(d)$, then we may compute its intersections with the event curves (see Fig. 7.8). These intersections correspond to critical directions that partition the curve $\theta = f(d)$ in regular subsets (open intervals and points). We need to compute only one backprojection per subset.

7.12 Conclusion

This chapter describes sound, complete, and polynomial planning algorithms in the presence of uncertainty. The key notion underlying these algorithms is that of a landmark area where control and position sensing are perfect. An important aspect of our algorithms, not investigated before, is that they allow uncertainty (more precisely, directional uncertainty) to vary over a continuous domain. Varying uncertainty has multiple applications. If the robot can control uncertainty (but with some cost associated with reduced uncertainty), the planner can try to generate a minimal-cost plan. If uncertainty is not controllable, varying uncertainty can be used to generate probabilistic plans or plans that can deal with unexpected obstacles.

The main technical result presented in this chapter is that planning does not require computing all the backprojections of the successive goal extensions, for all directions of motion d in S^1 and all values of the directional uncertainty θ in $[\theta_{min}, \theta_{max}]$. The $d\theta$-plane can be decomposed by critical curves into a polynomial collection of cells such that all backprojections in a cell are equivalent relative to planning. This result directly yields our planners.

In general, the task of a motion planner in the presence of uncertainty is threefold: (1) cover the set of configurations reachable by the robot by a finite collection of states, (2) synthesize functions able to recognize the achievement of these states, and (3) plan motion commands to move from state to state. The most important effect of introducing landmarks in the configuration space is to reduce the interdependence between these three activities. Actually, the states that the robot may attain are the landmark areas and the complement of their union. The recognition functions are also trivialized by the definition of the landmarks. Hence, planning is reduced to finding appropriate commands to move from landmark to landmark. To construct the termination sets of commands, the planner may have to consider any combination of states. In theory, there is

an exponential number of such combinations, but the algorithms presented in this chapter show that only a growing sequence of combinations (the successive goal extensions) must actually be considered. The length of this sequence is bounded by the number of landmark areas.

The creation of landmarks requires some prior engineering, which has some cost, and/or introduces some limitations. Actually, what the planner does not have to do, e.g., selecting a set of states and synthesizing state-recognition functions, is hidden either in this engineering work, or in the fact that our planner only solves a limited class of problems. However, as we saw in Section 7.9, it is possible to alleviate some of the assumptions underlying the definition of a landmark, thus extending the class of problems for which our planners remain useful.

Chapter 8

Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces

This chapter is based on the paper "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces," by L. Kavraki, P. Švestka, J.C. Latombe, and M. Overmars, which will appear in IEEE Tr. on Robotics and Automation.

8.1 Introduction

We present a new planning method which computes collision-free paths for robots of virtually any type moving among stationary obstacles (static workspaces). However, our method is particularly interesting for robots with many degrees of freedom (dof), say five or more. Indeed, an increasing number of practical problems involve such robots, while very few effective motion planning methods, if any, are available to solve them. The method proceeds in two phases: a *learning phase* and a *query phase.*

In the learning phase a probabilistic *roadmap* is constructed by repeatedly generating random free configurations of the robot and connecting these configurations using some simple, but very fast motion planner. We call this planner the *local planner.* The roadmap thus formed in the free configuration space (C-space [148]) of the robot is stored as an undirected graph R. The configurations are the nodes of R and the paths computed by the local planner are the edges of R. The learning phase is concluded by some postprocessing of R to improve its connectivity.

Following the learning phase, multiple queries can be answered. A *query* asks for a path between two free configurations of the robot. To process a query the method first attempts to find a path from the start and goal configurations to two nodes of the roadmap. Next, a graph search is done to find a sequence of edges connecting these nodes in the roadmap. Concatenation of the successive path segments transforms the sequence found into a feasible path for the robot.

Notice that the learning and the query phases do not have to be executed sequentially. Instead, they can be interwoven to adapt the size of the roadmap to difficulties encountered during the query phase, thus increasing the learning flavor of our method. For instance, a small roadmap could be first constructed; this roadmap could then be augmented (or reduced) using intermediate data generated while queries are being processed. This interesting possibility will not be explored in the chapter, though it is particularly useful to conduct trial-and-error experiments in order to decide how much computation time should be spent in the learning phase.

To run our planning method the values of several parameters must first be selected, e.g., the time to be spent in the learning phase. While these values depend on the *scene*, i.e., the robot and the workspace, it has been our experience that good results are obtained with values spanning rather large intervals. Thus, it is not difficult to choose one set of satisfactory values for a given scene or family of scenes, through some preliminary experiments. Moreover, increased efficiency can be achieved by tailoring several components of our method, in particular the local planner, to the considered robots. Overall, we found the method quite easy to implement and run. Many details can be engineered in one way or another to fit better the characteristics of an application domain.

We have demonstrated the power of our method by applying it to a number of difficult motion planning problems involving a variety of robots. In this chapter we report in detail on experiments with planar articulated robots (or linkages) with many dofs moving in constrained workspaces. However, the method is directly applicable to other kinds of holonomic robots, such as spatial articulated robots in 3D workspaces [120]. Additionally, a version of the method described here has been successfully applied to nonholonomic car-like robots [174]. In all cases, experimental results show that the learning times required for the construction of adequate roadmaps, i.e., roadmaps that capture well the connectivity of the free C-space, are short. They range from a few seconds[1] for relatively easy problems to a few minutes for the most difficult problems we have dealt with. Once a good roadmap has been constructed, planning queries are processed in a fraction of a second.

The very small query times make our planning method particularly suitable for many-dof robots performing several point-to-point motions in known static workspaces. Examples of tasks meeting these conditions include maintenance of cooling pipes in a nuclear plant, point-to-point welding in car assembly, and cleaning of airplane fuselages. In such tasks, many dofs are needed to achieve successive desired configurations of the end-effector while avoiding collisions of the rest of the arm with the complicated workspace. Explicit programming of such robots is tedious and time consuming. An efficient and reliable planner would considerably reduce the programming burden.

[1] All running times reported in this chapter have been obtained on a DEC Alpha workstation, except those given in Section 6 which were obtained with a Silicon Graphics Indigo workstation.

This chapter is organized as follows. Section 8.2 gives an overview of some previous research and relates our work to this research. Section 8.3 describes our motion planning method in general terms, i.e., without focusing on any specific type of holonomic robot. Both the learning and the query phases are discussed here in detail. Next, in Sections 8.4, 8.5, and 8.6 we apply our method to planar articulated robots. In Section 8.4 we describe specific techniques that can be substituted for general ones in the planner to handle these robots more efficiently (especially when these have many dofs). In Sections 8.5 and 8.6 we describe a number of experiments and their results; we also analyze how variations of some parameter values affect planning results. Section 8.5 presents results obtained with a customized implementation of the method embedding the specific techniques of Section 8.4. Section 8.6 discusses other experimental results obtained with a general implementation of the method. Section 8.7 concludes the chapter.

8.2 Relation to previous work

Path planning for robots in known and static workspaces has been studied extensively over the last two decades [130]. Recently there has been renewed interest in developing heuristic, but practical path planners. For few-dof robots, many such planners have been designed and some are extremely fast (e.g., [69, 140]). Considerable attention is now directed toward the creation of efficient heuristic planners for many-dof robots. Indeed, while such robots are becoming increasingly useful in industrial applications, complete methods for such robots have overwhelming complexity. New emerging applications also motivate that trend, e.g., computer graphic animation, where motion planning can drastically reduce the amount of data input by human animators, and molecular biology, where motion planning can be used to compute motions of molecules (modeled as spatial linkages with many dofs) docking against other molecules.

The complexity of complete path planning methods in high-dimensional configuration spaces has led researchers to seek heuristic methods that embed weaker notions of completeness (e.g., probabilistic completeness) and/or can be partially adapted to specific problem domains in order to boost performance in those domains.

In recent years, some of the most impressive results were obtained using potential field methods. Such methods are attractive, since the heuristic function guiding the search for a path, the potential field, can easily be adapted to the specific problem to be solved, in particular the obstacles and the goal configuration. The main disadvantage of these planners is the presence of local minima in the potential fields. These minima may be difficult to escape. Local minima-free potential functions (also called navigation functions) have been defined in [123, 169, 68]. But these functions are expensive to compute in high-dimensional configuration spaces and have not been used for many-dof robots.

One of the first successful potential field planners for robots with many dof is described in [101]. This planner has been used to compute paths of an 8-dof manipulator among vertical pipes in a nuclear plant, with interactive human assistance to escape local minima. In [102] the same authors

present a learning scheme to avoid falling into the local minima of the potential field. During the learning phase, probabilities of moving between neighboring configurations without falling into a local minimum are accumulated in an r^n array, where n is the number of dofs and r is the number of intervals discretizing the range of each dof. During the planning phase, these probabilities are used as another heuristic function (in addition to the potential function) to guide the robot away from the local minima. This learning scheme was applied with some success to robots with up to 6 dofs. However, the size of the r^n array becomes impractical when n grows larger.

Techniques for both computing potential functions and escaping local minima in high-dimensional C-spaces are presented in [68, 69]. The Randomized Path Planner (RPP) described in [68] escapes local minima by executing random walks. It has been successfully experimented on difficult problems involving robots with 3 to 31 dofs. It has also been used in practice with good results to plan motions for performing riveting operations on plane fuselages [108], and to plan disassembly operations for the maintenance of aircraft engines. Recently, RPP has been embedded in a larger "manipulation planner" to automatically animate scenes involving human figures modeled with 62 dofs [124]. However, several examples have also been identified where RPP behaves poorly [82, 182]. In these examples, RPP falls into local minima whose basins of attraction are mostly bounded by obstacles, with only narrow passages to escape. The probability that any random walk finds its way through such a passage is almost zero. In fact, once one knows how RPP computes the potential field, it is not too difficult to create such examples. One way to prevent this from happening is to let RPP randomly use several potential functions, but this solution is rather time consuming. Our roadmap planner deals efficiently with problems that are difficult for RPP, as discussed in Section 8.5.

Other interesting lines of work include the method in [70] which is based on a variational dynamic programming approach and can tackle problems of similar complexity to the problems solved by RPP. In [110, 111] a sequential framework with backtracking is proposed for serial manipulators, and in [87] a motion planner with performance proportional to task difficulty is developed for arbitrary many-dof robots operating in cluttered environments. The planner in [128] finds paths for six-dof manipulators using heuristic search techniques that limit the part of the C-space that is explored, and the planner in [64] utilizes genetic algorithms to help search for a path in high-dimensional C-spaces. Parallel processing techniques are investigated in [82, 147].

The planning method presented in this chapter differs significantly from the methods referenced above, which are for the most part based on potential field or cell decomposition approaches. Instead, our method applies a roadmap approach [130], that is, it constructs a network of paths in free C-space. Previous roadmap methods include the visibility graph [150], Voronoi diagram [156], and silhouette [79] methods. All these three methods compute in a single shot a roadmap that completely represents the connectivity of the free C-space. The visibility graph and Voronoi diagram methods are limited to low-dimensional C-spaces. The silhouette method applies to C-spaces of any dimension, but its complexity makes it little practical.

Roadmaps have also been built and used incrementally in several other planners. The planner in [81] incrementally builds the skeleton of the C-space using a local opportunistic strategy. This work

has inspired the approaches in [168, 88] which construct retracts of the free C-space using sensor data and thus do not assume that the (static) environment in which the robot moves is known a priori. The approach in [85] builds a sparse network of robot subgoals with the use of a simple and a computationally expensive planner. This network can also include information to accommodate local changes in the environment [86, 67].

Our method emphasizes efficiency and is primarily developed for robots with many dofs which move in static environments. We are not aware of other roadmap techniques that have been effectively applied to high-dimensional C-spaces. The approach we discuss in this chapter uses probabilistic techniques to incrementally build a roadmap in the free C-space of the robot. It can produce a roadmap in any amount of allocated time. If the time spent on the construction of the roadmap is short, the roadmap may not adequately represent the connectivity of the free C-space. Actually, in our planner, the roadmap is never guaranteed to fully represent free C-space connectivity, though if we let our techniques run long enough it eventually will (but we don't know how long is enough). However, while building the roadmap, our method heuristically identifies "difficult" regions in free C-space and generates additional configurations in those regions to increase network connectivity. Therefore, the final distribution of configurations in the roadmap is not uniform across free C-space; it is denser in regions considered difficult by the heuristic function. This feature helps to construct roadmaps of a reasonable size that represent free C-space connectivity well. In particular, it allows our implemented planner to efficiently solve tricky problems requiring proper choices among several narrow passages, i.e., the kind of problems that potential field techniques like RPP tackle poorly.

Note also that, like most practical methods for many-dof robots (one exception is the method in [FT89]), RPP is a one-shot method, i.e., it does not precompute any knowledge of the free C-space that is transferred from one run to another. Consequently, on problems that both RPP and our method solve well, the latter is usually much faster, once it has constructed a good roadmap. But, if the learning time is included in the duration of the path planning process (which should be the case whenever planning is done only once in a given workspace), there are many problems for which RPP is faster.

The work presented in this chapter is a cooperation between two different teams (Stanford University and University of Utrecht). It builds upon research previously done by these two teams. A single-shot random planner was described in [157] and was subsequently expanded into a learning approach in [158]. In these papers the emphasis was on robots with a rather low number of dofs. Similar techniques have been applied both to car-like robots that can move forward and backward (symmetrical nonholonomic robots) and car-like robots that can only move forward [173, 174]. In [175] these results are extended to simultaneous motion planning for multiple car-like robots. Independently, a preprocessing scheme similar to the learning phase was introduced in [118]. This scheme also builds a probabilistic roadmap in free C-space, but focuses on the case of many-dof robots. The need to expand the roadmap in "difficult" regions of C-space was noted there and addressed with simple techniques. Better expansion techniques were introduced in [119, 120]. That approach is described in detail in [117] and a theoretical analysis bounding the time spent by that planner is given in [121] and in [71]. The present chapter combines the ideas of the experimental work in these previous papers. Since it only presents a limited subset of the experimental results we

have obtained with our method, the interested reader is encouraged to look into our previous papers for additional results, in particular results involving other types of robots. Though computation times reported in these papers were obtained with different versions of our method, their orders of magnitude remain meaningful.

Finally, it should be noted that another planner which bares similarities with our approach, but was developed independently of our two teams, is proposed in [113].

8.3 The general method

We now describe our path planning method in general terms for a holonomic robot without focusing on any specific type of robot. During the learning phase a data structure called the roadmap is constructed in a probabilistic way for a given scene. The roadmap is an undirected graph $R = (N, E)$. The nodes in N are a set of configurations of the robot appropriately chosen over the free C-space. The edges in E correspond to (simple) paths; an edge (a, b) corresponds to a feasible path connecting the configurations a and b. These paths, which we refer to as local paths, are computed by an extremely fast, though not very powerful planner, called the local planner. The local paths are not explicitly stored in the roadmap, since recomputing them is very cheap. This saves considerable space, but requires the local planner to succeed and fail deterministically. We assume here that the learning phase is entirely performed before any path planning query. As we already noted, however, the learning and query phases can also be interwoven.

In the query phase, the roadmap is used to solve individual path planning problems in the input scene. Given a start configuration s and a goal configuration g, the method first tries to connect s and g to some two nodes $\tilde{s}$ and $\tilde{g}$ in N. If successful, it then searches R for a sequence of edges in E connecting $\tilde{s}$ to $\tilde{g}$. Finally, it transforms this sequence into a feasible path for the robot by recomputing the corresponding local paths and concatenating them.

In the following, we let $\mathcal{C}$ denote the robot's C-space and $\mathcal{C}_f$ its free subset (also called the free C-space).

8.3.1 The learning phase

The learning phase consists of two successive steps, which we refer to as the construction and the expansion step. The objective of the former is to obtain a reasonably connected graph, with enough vertices to provide a rather uniform covering of free C-space and to make sure that most "difficult" regions in this space contain at least a few nodes. The second step is aimed at further improving the connectivity of this graph. It selects nodes of R which, according to some heuristic evaluator, lie in difficult regions of C-space and expands the graph by generating additional nodes in their neighborhoods. Hence, the covering of $\mathcal{C}_f$ by the final roadmap is not uniform, but depends on the local intricacy of the C-space.

8.3.1.1 The construction step

Initially the graph $R = (N, E)$ is empty. Then, repeatedly, a random free configuration is generated and added to N. For every such new node c, we select a number of nodes from the current N and try to connect c to each of them using the local planner. Whenever this planner succeeds to compute a feasible path between c and a selected node n, the edge (c, n) is added to E. The actual local path is not memorized.

The selection of the nodes to which we try to connect c is done as follows. First, a set N_c of candidate neighbors is chosen from N. This set is made of nodes within a certain distance of c, for some metric D. Then we pick nodes from N_c in order of increasing distance from c. We try to connect c to each of the selected nodes if it is not already graph-connected to c. Hence, no cycles can be created and the resulting graph is a forest, i.e., a collection of trees. Since a query would never succeed *thanks to* an edge that is part of a cycle, it is indeed sensible not to consume time and space computing and storing such an edge. However, in some cases, the absence of cycles may lead the query phase to construct unnecessary long paths. This drawback can easily be eliminated by applying smoothing techniques to either the roadmap during the learning phase, or the particular paths constructed in the query phase, or both. Even if the roadmap contained cycles, such smoothing operations would eventually produce better paths.

Whenever the local planner succeeds in finding a path between two nodes, the connected components of R are dynamically updated. Therefore, no graph search is required for deciding whether a node picked from N_c is already connected to c, or not.

To make our presentation more precise, let:

- Δ be a symmetrical function $\mathcal{C}_f \times \mathcal{C}_f \rightarrow \{0, 1\}$, which returns whether the local planner can compute a path between the two configurations given as arguments;
- D be a function $\mathcal{C} \times \mathcal{C} \rightarrow R^+ \cup \{0\}$, called the *distance function*, defining a pseudo-metric in $\mathcal{C}$. (We only require that D be symmetrical and non-degenerate.)

The construction step algorithm can be outlined as follows:

(1) $N \leftarrow \emptyset$
(2) $E \leftarrow \emptyset$
(3) **loop**
(4) $\quad c \leftarrow$ a randomly chosen free configuration
(5) $\quad N_c \leftarrow$ a set of candidate neighbors of c chosen from N
(6) $\quad N \leftarrow N \cup \{c\}$
(7) $\quad$ **forall** $n \in N_c$, in order of increasing $D(c, n)$ **do**
(8) $\quad\quad$ **if** $\neg$*same_connected_component*$(c, n) \wedge \Delta(c, n)$ **then**
(9) $\quad\quad\quad E \leftarrow E \cup \{(c, n)\}$
(10) $\quad\quad\quad$ update R's connected components

A number of components of algorithm above are still unspecified. In particular, we need to define how random configurations are created in (4), propose a local planner for (8), clarify the notion of a candidate neighbor in (5), and choose the distance function D used in (7).

Creation of random configurations. The nodes of R should constitute a rather uniform random sampling of C_f. Every such configuration is obtained by drawing each of its coordinates from the interval of values of the corresponding dof using the uniform probability distribution over this interval. The obtained configuration is checked for collision. If it is collision-free, it is added to N; otherwise, it is discarded.

Collision checking requires testing if any part of the robot intersects an obstacle and if two distinct bodies of the robot intersect each other. It can be done using a variety of existing general techniques. In the general implementation considered in Section 8.6 the test is performed analytically using optimized routines from the PLAGEO library [107]. Alternatively, we could use an iterative collision checker, like the one described in [166], which automatically generates successive approximations of the objects involved in the collision test. In 2D workspaces, we may use a faster, but more specific collision checker (see Section 8.4).

The local planner. Our best experimental results have been obtained when the local planner is both deterministic and very fast. These requirements are not strict, however. We discuss briefly the tradeoffs involved in the choice of the local planner.

If a non-deterministic planner was used, local paths would have to be stored in the roadmap. The roadmap would require more space, but this would not be a major problem.

Concerning how fast the local planner should be, there is clearly a tradeoff between the time spent in each individual call of this planner and the number of calls. If a powerful local planner was used, it would often succeed in finding a path when one exists. Hence, relatively few nodes would be required to build a roadmap capturing the connectivity of the free C-space sufficiently well to reliably answer path planning queries. Such a local planner would probably be rather slow, but this could be somewhat compensated by the small number of calls needed. On the other hand, a very fast planner is likely to be less successful. It will require more configurations to be included in the roadmap; so, it will be called more often, but each call will be cheaper.

The choice of the local planner also affects the query phase. The purpose of having a learning phase is to make it possible to answer path planning queries quasi-instantaneously. It is thus important to be able to connect any given start and goal configurations to the roadmap, or to detect that no such connection is possible, very quickly. This requires that the roadmap be dense enough, so that it always contains a few nodes (at least one) to which it is easy to connect each of the start and goal configurations. It thus seems preferable to use a very fast local planner, even if it is not too powerful, and build large roadmaps with configurations widely distributed over the free C-space. In addition, if the local planner is very fast, we can use the same planner to connect the start and goal configurations to the roadmap at query time. Local paths needed not be memorized since

recomputing them at query time is inexpensive. We actually tried several local planners, some very fast, some slower but more powerful, and our experimental observations clearly confirmed this conclusion (e.g., see [154, 173]).

A quite general such local planner, which is applicable to all holonomic robots, connects any two given configurations by a straight line segment in configuration space and checks this line segment for collision and joint limits (if any). Verifying that a straight line segment remains within the joint limits is straightforward. On the other hand, collision checking can be done as follows [68]. First, discretize the line segment (more generally, any path generated by the local planner) into a number of configurations $c_1, \ldots, c_m$, such that for each pair of consecutive configurations (c_i, c_{i+1}) no point on the robot, when positioned at configuration c_i, lies further than some `eps` away from its position when the robot is at configuration c_{i+1} (`eps` is a predetermined positive constant).[2] Then, for each configuration c_i, test whether the robot, when positioned at c_i and "grown" by `eps`, is collision-free, using the collision checker discussed above. If none of the m configurations yield collision, conclude that the path is collision-free. Since `eps` is constant, the computation of the robot bodies grown by `eps` is done only once. In the following we will refer to this local planner as the *general local planner.*

The node neighbors. Another important choice to be made is that of the set N_c, the candidate neighbors of c. The local planner will be called to connect c with nodes in N_c and the cumulative cost of these invocations dominates learning time.

We avoid calls of the local planner that are likely to return failure by submitting only pairs of configurations whose relative distance (according to the distance function D) is smaller than some constant threshold `maxdist`. Thus, we define:

$$N_c = \{\tilde{c} \in N | D(c, \tilde{c}) \leq \texttt{maxdist}\}.$$

Additionally, according to the algorithm outline given above, we try to connect c to all nodes in N_c in order of increasing distance from c; but we skip those nodes which are in the same connected component as c at the time the connection is to be tried. By considering elements of N_c in this order we expect to maximize the chances of quickly connecting c to other configurations and, consequently, reduce the number of calls to the local planner (since every successful connection results in merging two connected components into one).

In our experiments we found it useful to bound the size of the set N_c by some constant `maxneighbors` (typically on the order of 30). This additional criterion guarantees that, in the worst case, the running time of each iteration of the main loop of the construction step algorithm is independent of the current size of the roadmap R. Thus, the number of calls to the local method is linear in the size of the graph it constructs.

[2] Throughout this chapter symbols in teletyped characters are used to denote parameters of the planning method.

The distance function. The function D is used to both construct and sort the set N_c of candidate neighbors of each new node c. It should be defined so that, for any pair (c, n) of configurations, $D(c, n)$ reflects the chance that the local planner will *fail* to compute a feasible path between these configurations. One possibility is to define $D(c, n)$ as a measure (area/volume) of the workspace region swept by the robot when it moves along the path computed by the local planner between c and n in the absence of obstacles. Thus, each local planner would automatically induce its own specific distance function. Since exact computation of swept areas/volumes tends to be rather time-consuming, a rough but inexpensive measure of the swept-region gives better practical results. For example, when the general local planner described above is used to connect c and n, $D(c, n)$ may be defined as follows:

$$D(c, n) = \max_{x \in \text{robot}} \|x(n) - x(c)\|, \tag{8.1}$$

where x denotes a point on the robot, $x(c)$ is the position of x in the workspace when the robot is at configuration c, and $\|x(n) - x(c)\|$ is the Euclidean distance between $x(c)$ and $x(n)$.

8.3.1.2 The expansion step

If the number of nodes generated during the construction step is large enough, the set N gives a fairly uniform covering of the free C-space. In easy scenes R is then well connected. But in more constrained ones where free C-space is actually connected, R often consists of a few large components and several small ones. It therefore does not effectively capture the connectivity of $\mathcal{C}_f$.

The expansion step is intended to improve the connectivity of the graph R generated by the construction step. Typically, if the graph is disconnected in a place where $\mathcal{C}_f$ is not, this place corresponds to some narrow, hence difficult region of the free C-space. The idea underlying the expansion step is to select a number of nodes from N which are likely to lie in such regions and to "expand" them. By expanding a configuration c, we mean selecting a new free configuration in the neighborhood of c, adding this configuration to N, and trying to connect it to other nodes of N in the same way as in the construction step. So, the expansion step increases the density of roadmap configurations in regions of $\mathcal{C}_f$ that are believed to be difficult. Since the "gaps" between components of the graph R are typically located in these regions, the connectivity of R is likely to increase.

We propose the following probabilistic scheme for the expansion step. With each node c in N we associate a positive weight $w(c)$ that is a heuristic measure of the "difficulty" of the region around c. Thus, $w(c)$ is large whenever c is considered to be in a difficult region. We normalize w so that all weights together (for all nodes in N) add up to one. Then, repeatedly, we select a node c from N with probability:

$$Pr(c \text{ is selected}) = w(c),$$

and we expand this node.

There are several ways to define the heuristic weight $w(c)$. One possibility is to count the number of nodes of N lying within some predefined distance of c. If this number is low, the obstacle region

probably occupies a large subset of c's neighborhood. This suggests that $w(c)$ could be defined inversely proportional to the number of nodes within some distance of c. Another possibility is to look at the distance d_c from c to the nearest connected component not containing c. If this distance is small, then c lies in a region where two components failed to connect, which indicates that this region might be a difficult one (it may also be actually obstructed). This idea leads to defining $w(c)$ inversely proportional to d_c. Alternatively, rather than using the structure of R to identify difficult regions, we could define $w(c)$ according to the behavior of the local planner. For example, if the local planner often failed to connect c to other nodes, this is also an indication that c lies in a difficult region. Which particular heuristic function should be used depends to some extent on the input scene. A more detailed discussion on expansion techniques can be found in [117]. For the framework of this chapter, the following function has produced good results:

- At the end of the construction step, for each node c, compute the failure ratio $r_f(c)$ defined by:

$$r_f(c) = \frac{f(c)}{n(c)+1},$$

  where $n(c)$ is total number of times the local planner tried to connect c to another node and $f(c)$ is the number of times it failed. (Note: Whenever the local planner fails to connect two nodes c and n, this failure is counted in *both* the failure ratios of c and n. In this way, the configurations that are included in N at the very beginning of the construction step get meaningful failure ratios.)

- At the beginning of the expansion step, for every node c in N compute $w(c)$ proportional to the failure ratio, but scaled appropriately so that all weights add up to one, i.e.:

$$w(c) = \frac{r_f(c)}{\sum_{a \in N} r_f(a)}.$$

To expand a node c, we compute a short random-bounce walk starting from c. For holonomic robots, a random-bounce walk consists of repeatedly picking at random a direction of motion in C-space and moving in this direction until an obstacle is hit. When a collision occurs, a new random direction is chosen. And so on. The final configuration n reached by the random-bounce walk and the edge (c, n) are inserted into R. Moreover, the path computed between c and n is explicitly stored, since it was generated by a non-deterministic technique. We also record the fact that n belongs to the same connected component as c. Then we try to connect n to the other connected components of the network in the same way as in the construction step. The expansion step thus never creates new components in R. At worst, it fails to reduce the number of components.

The weights $w(c)$ are computed only once at the beginning of the expansion step and are not modified when new nodes are added to R. Once the expansion step is over, all remaining small components of R, if any, are discarded. Here, a component is considered small if its number of nodes is less than some `mincomponent` percent (typically 0.01%) of the total number of nodes in N. The graph R after discarding the small components represents the roadmap that will be used during the query phase. It may contain one or several components.

Let T_L be the time allocated to the learning phase. Clearly, the range of adequate values for T_L depends on the scene, and these value should be determined experimentally for each new scene. If T_C is the time spent on the construction step and T_E is the time spent on the expansion step, we have found that a 2:1 ratio between these times, i.e, $T_C = 2T_L/3$ and $T_E = T_L/3$, gives good results over a large range of problems.

8.3.2 The query phase

During the query phase, paths are to be found between arbitrary input start and goal configurations, using the roadmap constructed in the learning phase. Assume for the moment that the free C-space is connected and that the roadmap consists of a single connected component R. Given a start configuration s and goal configuration g, we try to connect s and g to some two nodes of R, respectively $\tilde{s}$ and $\tilde{g}$, with feasible paths P_s and P_g. If this fails, the query fails. Otherwise, we compute a path P in R connecting $\tilde{s}$ to $\tilde{g}$. A feasible path from s to g is eventually constructed by concatenating P_s, the recomputed path corresponding to P, and P_g reversed. If one wishes, this path may be improved by running a smoothing algorithm on it. Smoothing techniques that can be used here include the one in [136], which selects random segments of the global path and tries to shortcut them by using the local planner, and the method in [72], which iteratively performs local geometric operations (i.e., cutting off triangle corners).

The main question is how to compute the paths P_s and P_g. The queries should preferably terminate quasi-instantaneously, so no expensive algorithm is desired here. Our strategy for connecting s to R is to consider the nodes in R in order of increasing distance from s (according to D) and try to connect s to each of them with the local planner, until one connection succeeds. We ignore nodes located further than `maxdist` away from s, because the chance of success of the local planner is very low. If all connection attempts fail, we perform one or more random-bounce walks, as described in Subsection 8.3.1.2. But, instead of adding the node at the end of each such random-bounce walk to the roadmap, we now try to connect it to R with the local planner. As soon as s is successfully connected to R, we apply the same procedure to connect g to R.

The reconstruction of a robot path from the sequence of nodes in P reduces to the concatenation of the paths that take the robot between adjacent nodes in P. Some of these paths have been produced by random-bounce walks during the learning phase and are stored in the relevant edges of R. Paths that correspond to connections that have been found during learning by the local planner are recomputed. The local planner is deterministic and it will produce the same path every time it is called with the same input configurations. Collisions need not be checked along the recomputed local paths if the local planner has the property that it aborts when a collision is detected: all intermediate configurations along the path have been checked for collision when the local path was first computed. An example of a planner having the above property is the straight-line planner of Section 8.3.1.1. If the local planner performs a certain (deterministic) action when a collision is detected, then collisions need to be checked along the recomputed path so that the same action can be repeated just after a collision is detected.

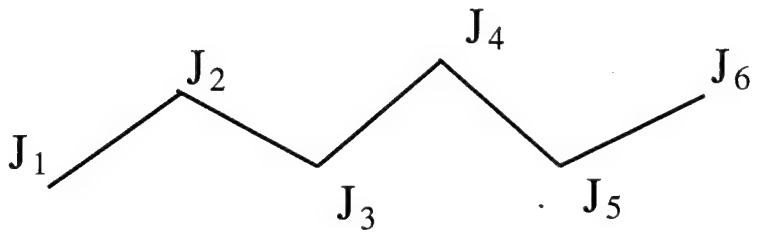


Figure 8.1: A planar articulated robot.

In general, however, the roadmap may consist of several connected components R_i, $i = 1, 2, \ldots, p$. This is usually the case when the free C-space is itself not connected. It may also happen when free C-space is connected, for instance if the roadmap is not dense enough. If the roadmap contains several components, we try to connect both s and g to two nodes in the *same* component, starting with the component closest to s and g. If the connection of s and g to some component R_i succeeds, a path is constructed as in the single-component case. The method returns failure if it cannot connect both the start and goal configuration to the same roadmap component. Since in most examples the roadmap consists of rather few components, failure is rapidly detected.

If path planning queries fail frequently, this is an indication that the roadmap may not adequately capture the connectivity of the free C-space. Hence, more time should be spent in the learning phase, i.e., $\mathtt{T_L}$ should be increased. However, it is not necessary to construct a new roadmap from the beginning. Since the learning phase is incremental, we can simply extend the current roadmap by resuming the construction step algorithm and/or the expansion step algorithm, starting with the current roadmap graph, thus interweaving the learning and the query phases.

8.4 Application to planar articulated robots

This section describes the application of our planning method to planar articulated robots with fixed or free bases. We present techniques specific to these robots that can be substituted for the more general techniques in the roadmap method in order to increase its efficiency. The purpose of this presentation is to illustrate the ease with which the general method for holonomic robots can be engineered to better suit the needs of a particular application. In Section 8.5 we will discuss experiments with an implementation of the method that embeds the specific techniques described below, while in Section 8.6 we will present experimental results with a general implementation of the method to demonstrate that the method remains quite powerful, even without specific components. In the rest of the chapter we will refer to these two implementations as the *customized implementation* and the *general implementation*, respectively.

To make the following presentation shorter, we only consider the following type of planar articulated

robots with an arbitrary number of revolute joints. Figure ?? illustrates such a robot in which the links are line segments. The links, which may actually be any polygons, are denoted by L_1 through L_q (in the figure, $q = 5$). Points J_1 through J_q designate revolute joints. Point J_1 denotes the base of the robot; it may, or may not, be fixed relative to the workspace. The point J_{q+1} (J_6 in the figure) is called the endpoint of the robot. Each revolute joint J_i ($i = 1, \ldots, q$) has defined certain internal joint limits, denoted by low_i and up_i, with $low_i < up_i$, which constrain the range of the possible orientations that L_i can take relative to L_{i-1}. If the robot's base is free, the translation of J_1 is bounded along the x and y axes of the Cartesian coordinate system embedded in the workspace by low_x and up_x, and low_y and up_y, respectively. We represent the C-space of such a q-link planar articulated robot by $[low_1, up_1] \times \ldots \times [low_q, up_q]$, if its base is fixed, and by $[low_x, up_x] \times [low_y, up_y] \times [0, 2\pi] \times [low_2, up_2] \times \ldots \times [low_q, up_q]$, if its base is free. A self-collision configuration is any configuration where two non-adjacent links of the robot intersect each other. We do not allow such configurations. Thus, the free C-space is constrained by the obstacles and by the set of self-collision configurations. We assume that the joint limits prevent self-collisions between any two adjacent links.

We now discuss specific techniques for local path planning, distance computation, and collision checking that apply well to the family of robots defined above. The same techniques can also be applied, possibly with minor adaptations, to other types of articulated robots, e.g., robots with prismatic joints and/or with multiple kinematic chains [119] and articulated robots in 3D workspace [120].

Local path planning. Let a and b be any two given configurations that we wish to connect with the local planner. Our local planner constructs a path as follows. It translates at constant relative velocity all the joints with an odd index, i.e., all J_{2*i+1}'s, along the straight lines in the workspace that connect their positions at configuration a to their positions at configuration b. During this motion the planner adjusts the position of every other joint J_{2*i} using the straightforward inverse kinematic equations of this point relative to J_{2*i-1} and J_{2*i+1}. Thus, the J_{2*i}'s "follow" the motion led by the J_{2*i+1}'s. If q is even, the position of J_q is not determined by the above rule; it is computed by rotating joint J_q at constant revolute velocity relative to the linear velocity of point J_q. Recall from Subsection 8.3.1.1 that a local path is discretized into a sequence of configurations for collision checking. When our specific technique is used, we must also verify that the coordinates of each such configuration are within joint limits. Thus, the motion is aborted if either a collision occurs, or a joint moves beyond one of its limits, or some J_{2*i} cannot follow the motion led by the J_{2*i+1}'s. We have observed that in cases when the above motion does not manage to connect configurations a and b, it nevertheless brings the robot to a configuration b' very close to b. It then pays off to try to connect b' and b with a straight line in C-space and only after this fails to declare failure of the local planner to connect a and b. We will refer to the above planner as the *specific local planner*.

The workspace region swept out by the robot along a local path computed by the specific local planner between two configurations a and b is typically smaller than for the path joining a and b by a straight line segment in configuration space, which is computed by the general local planner

described in Subsection 8.3.1.1. Hence, the local paths generated by the specific planner are more likely to be collision-free than those generated by the general planner. Also, collision checking is less expensive since, for a given **eps**, the discretization of the local path yields less configurations. On the other hand, the specific planner, though still very fast, is not as fast as the general planner. Indeed, it requires inverse kinematic computation to determine configuration coordinates along the path. Nevertheless, on examples involving many-dof planar articulated robots, better results are obtained when the specific local planner is used.

Distance computation. Let $J_i(a)$, $i = 1, \ldots, q+1$ denote the position of the point J_i in the workspace, when the robot is at configuration a. We define the distance function D by:

$$(a, b) \in \mathcal{C} \times \mathcal{C} \quad \mapsto \quad D(a, b) = \sum_{i=1}^{q+1} \|J_i(a) - J_i(b)\|^2,$$

where $\|J_i(x) - J_i(y)\|$ is the Euclidean distance between $J_i(a)$ and $J_i(b)$. This function is a better approximation of the area swept by the robot along the local paths computed by the specific local planner than the general distance function defined by Eqn. (8.1).

Collision checking. The 2D workspace allows for a very fast collision checking technique. In this technique each link of the robot is regarded as a distinct robot with two dofs of translation and one dof of rotation. A bitmap representing the 3D configuration space of this robot is precomputed, with the "0"'s describing the free subset of this space and the "1"'s describing the subset where the link collides with an obstacle. When a configuration is checked for collision, the 3D configuration of each link is computed and tested against its C-space bitmap, which is a constant-time operation. Different 3D bitmaps must be computed for links of different shape. However, if larger links can be modeled as two (or more) smaller links, then we need not create one bitmap for each link of the robot. For example, when all the links are line segments (as in Figure **??**), a single bitmap can be computed, for the shortest link. Then collision checking for a long link requires multiple access to the bitmap of the short link. The computation of any 3D bitmaps needed for collision checking is performed only once, prior to the learning phase.

The 3D bitmap for one link can be computed as a collection of 2D bitmaps, each corresponding to a fixed orientation of the link. If the link and the obstacles are modeled as collections of possibly overlapping convex polygons, the construction of a 2D bitmap can be done as follows [140]. First use the algorithm in [148] to produce the vertices of the obstacles in the link's C-space. (This algorithm takes linear time in the number of vertices of the objects.) Then draw and fill the obstacles into the 2D bitmap. (On many workstations, this second operation can be done very quickly using raster-scan hardware originally designed to efficiently display filled polygons on graphic terminals.) Each 2D bitmap may also be computed using the FFT-based method described in [116], whose complexity depends only on the size of the bitmap. This FFT method is advantageous when the obstacles are originally input as bitmaps. In any case, experiments show that computing a 3D

bitmap with a size on the order of $128 \times 128 \times 128$ takes a few seconds. Clearly, this technique is not yet practical for 3D workspaces, since it requires the generation of 6D bitmaps.

There are many other ways of adjusting our general path planning method to a specific robot. However, too much specific tuning may not always be desirable: at some point the gain in efficiency becomes smaller than the burden of making the specific changes and keeping track of them.

8.5 Results with customized implementation

In this section we present the performance of our method when this is implemented with the local planner, the collision checker, and the distance function described in Section 8.4. To be precise, while collision checking with obstacles is done using the bitmap technique, self-collisions are detected analytically.

The planner is implemented in C and for the experiments reported here we used a DEC Alpha workstation. This machine is rated at 126.0 SPECfp92 and 74.3 SPECint92. We have tested our planner on a number of 2D scenes. Each scene contains polygonal obstacles and a planar articulated robot whose links are line segments (see Figures 8.2 and 8.6). By no means does this reflect a limitation of the method. The specific local planner and collision checker of Section 8.4 also apply to robots made of polygonal links (though several bitmaps may then be required). The parameters of our planner are:

- T_C, the time to be spent in the construction step;
- T_E, the time to be spent in the expansion step;
- `maxdist`, the maximal distance between nodes that the local planner may try to connect;
- `eps`, the constant used to discretize local paths before collision checking;
- `maxneighbors`, the maximum number of calls of the local planner per node;
- $T_{\texttt{RB\_expand}}$, the duration of the computation of a random-bounce walk performed during the expansion step (learning phase);
- $N_{\texttt{RB\_query}}$, the maximum number of random-bounce walks allowed for connecting the start or goal configuration to the roadmap (query phase);
- $T_{\texttt{RB\_query}}$, the duration of the computation of each of the random-bounce walks during the query phase.

(Notice that the last two parameters determine an upper bound on the time spent in answering a query.)

For each scene, we first input a set of configurations by hand, which we refer to as the test set. For a fixed T_C and T_E, we then independently create many different roadmaps starting with different values of the random value generator. In the examples here we only keep the largest connected component of the roadmap; other components, if any, are simply discarded. We then try to connect the same configuration in the test set to each of these roadmaps and we record the percentage of times our planner succeeds to make a connection in a prespecified amount of time (2.5 seconds). The estimated success rates may be used to calculate the success rates of queries that involve any two configurations in the test set. By performing a large number of experiments, we believe that we present a realistic characterization of the performance of our planner. In particular, we ensure that the results do not reflect just a lucky run, or a bad one. We independently repeat the same experiment for different T_C and T_E. For the other parameters described above, we choose fixed values throughout the experiments based on some preliminary experimental results. Notice that it is important to choose the configurations in the test set manually. For obvious reasons, a random generation similar to the one used during the learning phase tends to produce configurations that are easily connected to the roadmap. Instead, proceeding manually allows us to select "interesting" configurations, for example configurations where the robot lies in narrow passages between workspace obstacles. It is unlikely that the random generator of the learning phase produced many such configurations.

We present results obtained with two representative scenes shown in Figures 8.2 (fixed-base robot) and 8.6 (free-base robot).

Fixed-base articulated robot. Figure 8.2 shows eight configurations forming the test set of an articulated robot in a scene with several narrow gates. The robot has a fixed base, denoted by a square, and 7 revolute degrees of freedom.

The table in Figure 8.3 reports the success rates of connecting the configurations in the test set to roadmaps obtained with different learning times. The learning time, T_L, is shown in column 1. It is broken into T_C and T_E in columns 2 and 3, with $T_E = T_C/2$. The values of the other parameters of the planner are: `maxdist` = 0.4, `eps` = 0.01 (for the interpretation of these two values note that the workspace is described as a unit square), `maxneighbors` = 30, $T_{\texttt{RB\_expand}}$ = 0.01 sec, $T_{\texttt{RB\_query}}$ = 0.05 sec, $N_{\texttt{RB\_query}}$ = 45.

For every row of the table in Figure 8.3 we independently generated 30 roadmaps, each with the indicated learning time. The roadmaps generated for different rows were also computed independently, that is, no roadmap in some row was reused to construct a larger one in following row.

Column 4 in Figure 8.3 gives the average number of collision checks performed for the roadmap construction for different learning times. This number can be regarded as an estimate of the computational complexity of the planner. In the context of the our approach, it must be interpreted with caution. Collision checks are done not only along robot paths, as in most planners, but also when trying to guess random free configurations of the robot (see Subsection 3.1.1). Most of these randomly guessed configurations are illegal because of collisions with the obstacles or self-intersections. On the average, this is quickly detected and the collision checker aborts almost

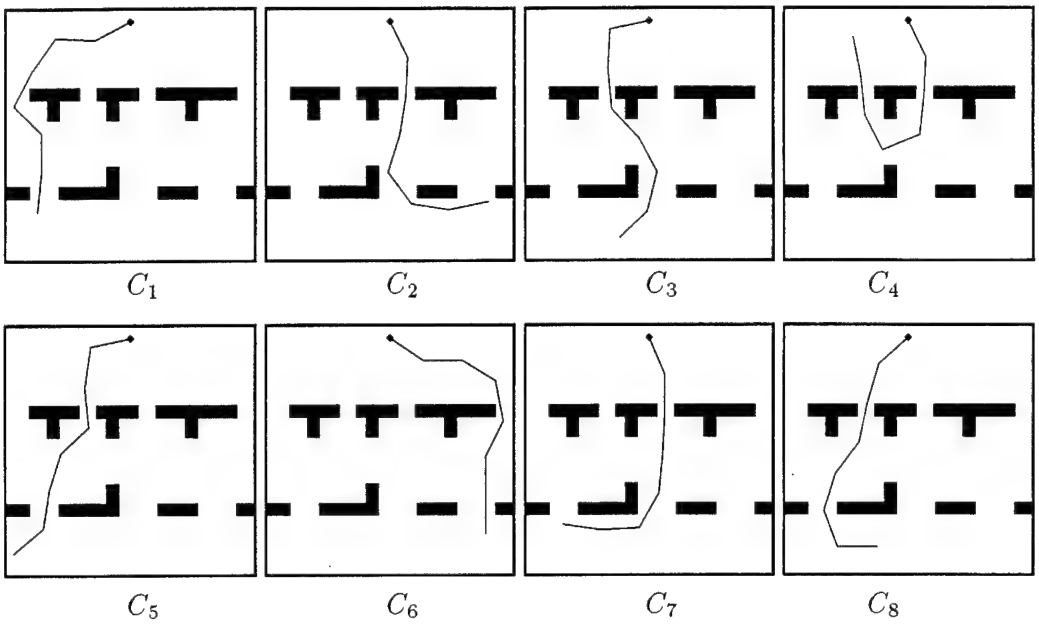


Figure 8.2: Scene 1, with 7-revolute-joint fixed-base robot.

T_L (sec)	T_C (sec)	T_E (sec)	Coll. checks	Avg. nodes	Success Rate (%) C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
20.1	13.1	7.0	621943	1062	100.0	36.7	56.7	36.7	53.3	100.0	36.7	60.0
30.1	19.5	10.6	889384	1643	100.0	66.7	70.0	66.7	76.7	100.0	66.7	80.0
40.3	26.3	14.0	1145091	2233	100.0	90.0	86.7	90.0	86.7	100.0	90.0	86.7
50.3	32.7	17.6	1392454	2783	100.0	96.7	96.7	96.7	96.7	100.0	96.7	96.7
60.2	39.1	21.1	1631612	3284	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
70.3	45.8	24.5	1876006	3805	100.0	96.7	100.0	96.7	100.0	100.0	96.7	100.0
80.4	52.2	28.2	2104209	4272	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

Figure 8.3: Results with customized planner for scene of Fig. 2 (with expansion).

T_L (sec)	T_C (sec)	T_E (sec)	Coll. checks	Avg. nodes	Success Rate (%) C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
20.0	20.0	0.0	597559	1011	100.0	13.3	36.7	10.0	40.0	93.3	13.3	36.7
30.1	30.1	0.0	852038	1601	100.0	50.0	46.7	46.7	46.7	90.0	53.3	46.7
40.2	40.2	0.0	1086053	2300	100.0	80.0	80.0	80.0	80.0	100.0	80.0	80.0
50.2	50.2	0.0	1291216	2877	100.0	90.0	96.7	90.0	96.7	100.0	90.0	96.7
60.2	60.2	0.0	1502089	3372	100.0	90.0	100.0	90.0	100.0	100.0	90.0	100.0
70.2	70.2	0.0	1688544	3877	100.0	96.7	100.0	96.7	100.0	100.0	96.7	100.0
80.3	80.3	0.0	1860341	4295	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

Figure 8.4: Results with customized planner for scene of Fig. 2 (no expansion).

T_L (sec)	T_C (sec)	T_E (sec)	Coll. checks	Size of components	Coll. checks for connection to roadmap							
					C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
20.3	13.3	7.0	620238	878, 116, 62	62	F	F	F	F	F	F	F
30.2	19.7	10.5	905312	1644, 165	78	51	F	7584	F	40	59	F
40.4	26.3	14.1	1178494	2411	53	1148	22	3432	33	44	225	2270
50.3	32.8	17.5	1421185	2881, 63, 10	13	20	20	3877	80	38	20	2328
60.4	39.3	21.1	1661916	3302, 35, 33	57	45	16	22	14	160	51	46
70.2	45.6	24.6	1917744	3869, 52, 10	94	30	19	4764	21	42	74	63
80.2	52.1	28.1	2128273	4245, 49	32	25	16	32	12	89	48	43

Figure 8.5: Connecting configurations to the roadmap.

instantaneously. Column 5 in Figure 8.3 reports the average number of nodes, over the 30 runs, in the largest roadmap component at the end of the learning phase. The largest connected component of each roadmap is used for query processing. Columns 6 though 13 are labeled with the eight configurations $C_1, \ldots, C_8$ of Figure 8.2. The columns report the success rate when trying to connect, in less that 2.5 seconds, the corresponding configuration to each of the 30 produced roadmaps. One trial (as defined by the parameters `maxdist`, `maxneighbors`, T_{RB_query}, and N_{RB_query}) was made per roadmap.

The table in Figure 8.3 shows that after a learning time of 60 seconds or more (rows 5, 6, and 7), all eight configurations of Figure 8.2 are successfully connected to the generated roadmaps with very few exceptions. These are all located in row 6, where configurations C_3, C_4 and C_7 were not connected to the roadmaps, once out of the 30 trials of that row. Such exceptions are to be expected with a randomized technique.

Figure 8.4 shows the percentage of successful connections to roadmaps created without expansion. The corresponding rows of the tables in Figures 8.3 and 8.4 report results obtained with the same learning time. We again generated 30 independent roadmaps in each row in Figure 8.4. We show the average number of collision checks required to create each roadmap (column 4), the average number of nodes in the largest component of these roadmaps (column 5), and the success rate when trying to connect $C_1, \ldots, C_8$ to them. In general, the percentages of successful connections are lower in this table. The difference shows more clearly when the learning time is small. If we are interested in obtaining a solution to a path planning problem as fast as possible, it is thus better to spend part of the time allocated to the learning phase on the expansion step rather than spend it completely on the construction step. As mentioned above, the ratio $T_C/T_E = 2$ gives good results over a wide range of problems.

Let us finally note that connecting $C_1, \ldots, C_8$ to the roadmaps is very fast, which in turn means that finding a path between any two of the above configurations is also fast. In Figure 8.5 we repeat the experiment of Figure 8.3, but now we create only one roadmap in each row of the table. We report in columns 6 to 13 the actual number of collision checks needed to connect $C_1, \ldots, C_8$ to the roadmaps produced after learning times of 20, 30, 40, 50, 60, 70 and 80 seconds. Again, we try to connect the configurations in the test set only to the largest component of these roadmaps,

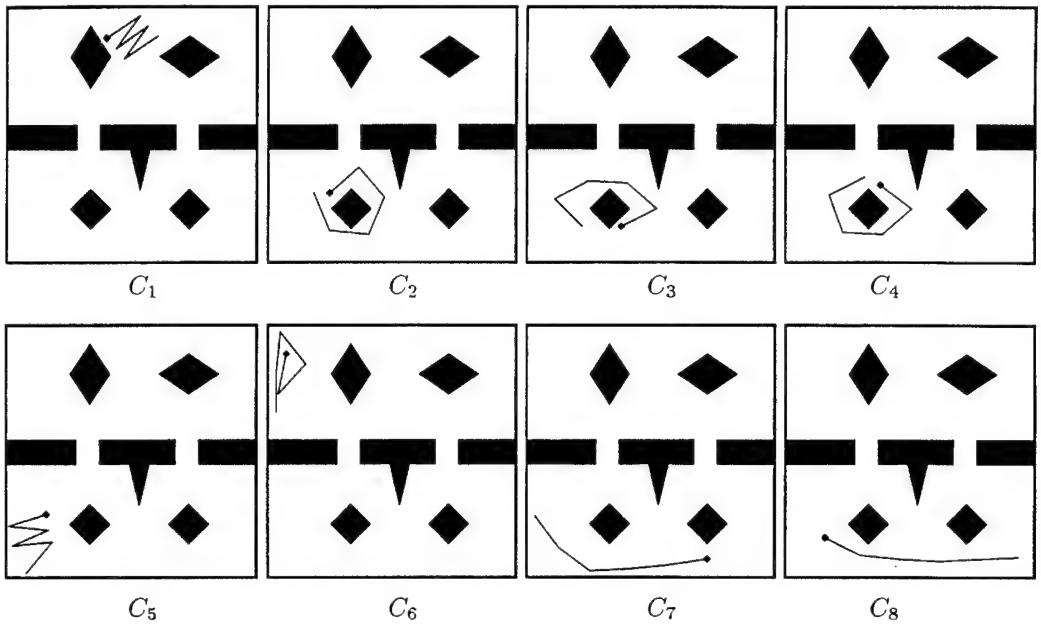


Figure 8.6: Scene 2, with 5-revolute-joint free-base robot (7-dof).

T_L (sec)	T_C (sec)	T_E (sec)	Coll. checks	Avg. nodes	Success Rate (%) C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
20.1	13.0	7.1	712661	541	96.7	6.7	6.7	6.7	6.7	86.7	6.7	10.0
30.1	19.6	10.5	1037739	924	100.0	16.7	16.7	16.7	16.7	90.0	16.7	16.7
40.1	26.0	14.1	1361134	1603	100.0	60.0	63.3	56.7	56.7	96.7	56.7	56.7
50.2	32.6	17.6	1674144	2460	100.0	93.3	93.3	93.3	93.3	100.0	96.7	93.3
60.3	39.2	21.1	1987967	2999	100.0	93.3	93.3	93.3	93.3	100.0	93.3	93.3
70.1	45.6	24.5	2336917	3695	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
80.4	52.3	28.1	2632712	4229	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

Figure 8.7: Results with customized planner for scene of Fig. 6 (with expansion).

T_L (sec)	T_C (sec)	T_E (sec)	Coll. checks	Avg. nodes	Success Rate (%) C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
20.0	20.0	0.0	686580	527	96.7	3.3	3.3	3.3	3.3	86.7	3.3	3.3
30.0	30.0	0.0	987852	1005	100.0	30.0	30.0	26.7	30.0	96.7	30.0	30.0
40.3	40.3	0.0	1265245	1437	100.0	40.0	40.0	40.0	43.3	100.0	43.3	40.0
50.1	50.1	0.0	1534808	2238	100.0	80.0	80.0	76.7	76.7	100.0	76.7	76.7
60.0	60.0	0.0	1778678	2709	100.0	80.0	80.0	80.0	80.0	100.0	83.3	80.0
70.0	70.0	0.0	2058469	3384	100.0	90.0	90.0	90.0	90.0	100.0	90.0	90.0
80.2	80.2	0.0	2277226	4002	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

Figure 8.8: Results with customized planner for scene of Fig. 6 (no expansion).

and we report failure (indicated by 'F') if we do not succeed to do so within the allocated time (2.5 seconds). In most cases, relatively few collision checks are needed for successful connection to the roadmap: a few tens to a few hundreds. Infrequently, a couple of thousands of collision checks are performed. This happens when one or more random-bounce walks are executed before the configuration is connected to the roadmap with the local planner. In any case, for the machine used for our experiments, the above numbers translate to connection times of a fraction of second to a few seconds. In the table of Figure 8.5 we report in column 4 the size of all roadmap components with more than 10 nodes. It is easy to see that after a learning time of 40 seconds, there is a clear difference in the size of the major component and the smaller ones. The latter contain only a small percentage of the total nodes and their presence does not affect path planning. That is why in our analysis we considered only the largest component of the roadmaps produced by learning.

Path planning will succeed between any two configurations that can be connected to the roadmaps produced. A simple breadth-first search algorithm can produce a sequence of edges that connect two nodes of the roadmap in a very short time, typically a small fraction of second in the machine used. Reconstructing the path is equally fast if no collisions need to be performed along recomputed local paths (see Section 8.3.2). In our implementation where collision checks are performed when recovering a local path, we spend a few tens of thousands of collision checks for connecting between different nodes in the roadmap. It is interesting to contrast the number of collision checks needed for learning and for query processing: collision checks for learning are 2 to 3 orders of magnitude larger than collision checks needed for answering queries. This is also true for any configurations we tried in the scene of Figure 8.2 and not only the eight configurations considered here. RPP, one of the few planners that can tackle the path planning problems arising from the configurations in Figure 8.2, takes a few tens of minutes on the average to solve these queries. Thus, even if learning time is included in the duration of the path planning process, our roadmap technique is still faster than RPP for this example. However, the above scene is very difficult for potential field methods. In simpler cases (see Section 8.6) RPP is equally fast, if not faster than our roadmap method.

Free-base articulated robot. We have performed the same experiments for a free-base articulated robot (see Figure 8.6). The robot has a total of 7 dof: 2 for its free base and 5 for its revolute joints. The parameter values of our planner are the same as in the previous experiments.

Figures 8.7 and 8.8 show the results obtained with and without expansion, respectively. 30 roadmaps were created independently for each row in the above tables. Again, in almost all cases, the percentage of successful connections to the roadmaps is greater with expansion than without (for the same total learning time). After a learning phase of 70 seconds, all configurations can be connected to the roadmaps produced. The actual number of collision checks for connecting $C_1, \ldots, C_8$ of Figure 8.6 to the roadmaps are again in the order of a few tens to a few thousands. This makes path planning between any two of the configurations shown in Figure 8.6 very fast: usually a fraction of a second in the machine used.

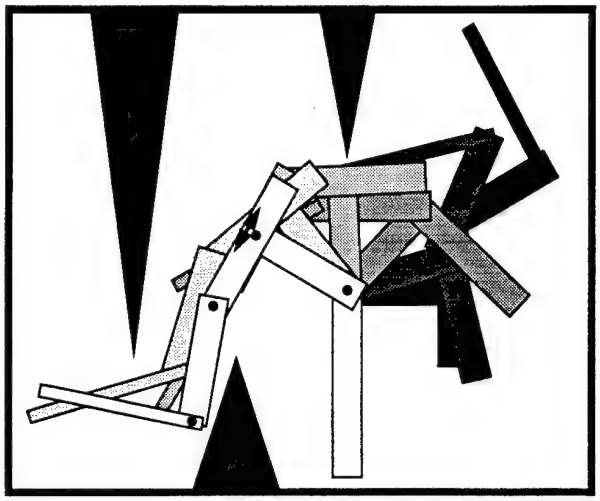


Figure 8.9: Scene 1, with 4-dof robot.

8.6 Results with general implementation

The customized implementation used in the previous section solves efficiently path planning problems involving planar articulated robots. In this section we demonstrate that the general implementation of the planner still gives very good results for a variety of examples.

The planner considered here is essentially an implementation of the method described in Section 8.3. Unlike the customized implementation, this implementation does not use any specific techniques for local path planning, collision checking, or distance computation. Hence, as described in Section 8.3, the local path constructed between any two configurations is the straight line segment joining them in C-space; the distance function D is the one defined by Eqn. (8.1); and collision checking is done analytically, using routines from the PLAGEO library [107]. The planner was implemented on a Silicon Graphics Indigo^2 workstation rated at 96.5 SPECfp92 and 90.4 SPECint92. This machine is comparable to the one we used for the results in the previous section. We report here on experimentation conducted with articulated robots with 4 or 5 joints connected by polygonal links. This general planner is directly applicable to robots with polyhedral links moving in 3D workspaces.

We present results obtained with two representative examples. In scene 1 in Figure **??**, we have a 4-dof robot with three revolute joints and one prismatic joint (indicated by the double arrow). Scene 2 in Figure **??**, is a slightly more difficult one, with a 5-revolute-joint robot and narrow areas in the workspace. For most existing planners, motion planning problems in both these scenes would be challenging ones. RPP is able to deal with these examples efficiently. Still, the cases treated here are considerably easier than in the scenes of Section 8.5, due to the relatively low number of dofs of the two robots, and the presence of only few tight areas in the workspaces.

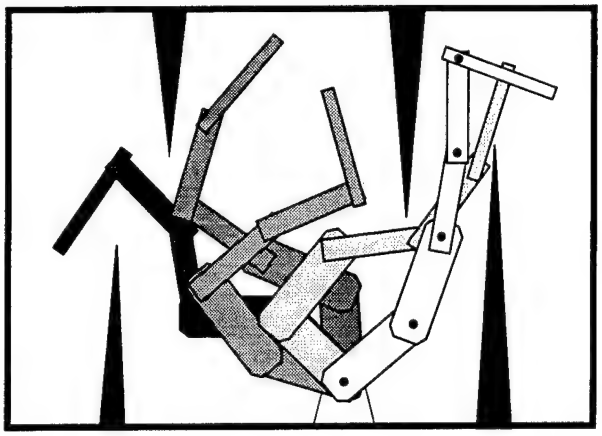


Figure 8.10: Scene 2, with 5-dof robot.

T_L (sec)	T_C (sec)	T_E (sec)	Coll. checks Learning: Scene 1	Success rate (%) Scene 1	Coll. checks Learning: Scene 2	Success rate (%) Scene 2
2.5	1.67	0.83	10078	53.3	9558	50
5	3.33	1.67	19756	93.3	18746	87
7.5	5	2.5	29525	100	27607	97
10	6.67	3.33	38607	100	36392	100

Figure 8.11: Results with general planner for scenes of Fig. 9 and 10.

T_L (min)	T_C (min)	T_E (min)	Coll. checks	Success Rate (%) C_1	C_4	C_7	C_8
5	3.3	1.7	700000	80.0	30.0	30.0	43.3
10	6.7	3.3	1000000	96.7	76.7	70.0	53.3
15	10	5.0	1500000	96.7	80.0	73.3	90.0
20	13.3	6.7	1800000	100.0	96.7	96.7	100.0
25	16.7	8.3	2000000	100.0	100.0	100.0	100.0

Figure 8.12: Results with general planner for scene of Fig. 2 (with expansion).

The experiments conducted with these two test scenes are similar to those in Section 8.5. For each scene, we consider only two "difficult" configurations s and g. Then, for a fixed construction time T_C and expansion time T_E (hence, a fixed learning time T_L), we independently create 30 roadmaps. For each of these roadmaps we only consider its main connected component and we test whether the query with configurations (s, g) succeeds within 2.5 seconds. In other words, we test whether *both* s and g can be quickly connected to the main connected component of the roadmap with the method described in Section 8.3.2. We repeat this experiment for a number of different construction times T_C and expansion times T_E, with $T_E = T_C/2$. For each such pair of times we report the success rate in answering the query (s, g).

The other parameters have the following fixed values, which are almost the same as in the experimentation reported in the previous section: `maxdist` = 0.5, `eps` = 0.01, `maxneighbors` = 30, $T_{\texttt{RB\_expand}} = 0.01$, $T_{\texttt{RB\_query}} = 0.05$ sec, and $N_{\texttt{RB\_query}} = 45$. Again, for the interpretation of the values for `maxdist` and `eps`, note that we scaled the two scenes in a way that the workspace obstacles just fit into the unit square.

In both Figures ?? and ?? the start configuration s is shown in dark grey, and the goal configuration g in white. In each figure, several robot configurations along a path solving the query are displayed using various grey levels. The results of the experiments described above are given in Figure 8.11. The average number of collision checks required to build the roadmaps is given in column 4 for scene 1 and in column 6 for scene 2. The query in scene 1 is solved in all 30 cases after having learned for 7.5 seconds. Learning for 5 seconds though suffices to successfully answer the query in more than 90% of the cases. In scene 2 we observe a similar behavior, although the required learning times are slightly higher.

These results show that the general implementation is able to efficiently solve rather complicated planning problems. However, when applied to problems involving more dofs, like those in the previous section, the learning times required to build good roadmaps are much longer. For example, experiments indicated that about 25 minutes of learning are required in order to obtain roadmaps that capture well the free C-space connectivity of the scene shown in Figure 8.2. Figure 8.12 reports some experimental results obtained over many independently constructed roadmaps, for different learning times. As in Section 8.5, we estimate the average number of collision checks needed

during learning and the percentage of times that our planner succeeds in connecting some of the configurations of Figure 8.2 to the roadmap, over many independently constructed roadmaps, for different learning times. In such difficult cases, clearly, customization is desirable, if not necessary.

8.7 Conclusion

We have described a two-phase method to solve robot motion planning problems in static workspaces. In the learning phase, the method constructs a probabilistic roadmap as a collection of configurations randomly selected across the free C-space. In the query phase, it uses this roadmap to quickly process path planning queries, each specified by a pair of configurations. The learning phase includes a heuristic evaluator to identify difficult regions in the free C-space and increase the density of the roadmap in those regions. This feature enables us to construct roadmaps that capture well the connectivity of the free C-space.

The method is general and can be applied to virtually any type of holonomic robot. Furthermore, it can be easily customized to run more efficiently on some family of problems. Customization consists of replacing components of the general method, such as the local planner, by more specialized ones fitting better the characteristics of the considered scenes. In this chapter, we have described techniques to customize the method to planar articulated robots, and presented experimental results with both a general and a customized implementation of the method. The customized implementation can solve very difficult path planning queries involving many-dof robots in a fraction of a second, after a learning time of a few dozen seconds. The general implementation solves the same problems in several minutes, but it is still very efficient in less difficult problems.

In [119, 120, 117, 158] prior versions of the method have been applied to a great variety of holonomic robots including planar and spatial articulated robots with revolute, prismatic, and/or spherical joints, fixed or free base, and single or multiple kinematic chains. In [173, 174, 175] a variation of the method (essentially one with a different general local planner) was also run successfully on examples involving nonholonomic car-like robots.

Experimental results show that our method can efficiently solve certain kinds of problems which are beyond the capabilities of other existing methods. For example, for planar articulated robots with many dofs, the customized implementation of Section 8.5 is much more consistent than the Randomized Path Planner (RPP) of [68]. Indeed, the latter can be very fast on some difficult problems, but it may also take prohibitive time on some others. We have not observed such disparity with our roadmap method. Moreover, after sufficient learning (usually on the order of a few dozen seconds), the probabilistic roadmap method answers queries considerably faster than RPP. However, when the learning time is included in the planning time, RPP is faster on many problems, since it does not perform any substantial precomputation.

An important question is how our method scales up when we consider scenes with more complicated geometry, since the cost of collision checking is much higher. First, let us note that in 2D workspaces the effect is likely to be limited if the bitmap collision-checking technique of Section 8.4 is used.

Indeed, once bitmaps have been precomputed, collision checking is a constant-time operation; and the cost of computing bitmaps using the FFT-based technique described in [116] only depends on the resolution (i.e., the size) of these bitmaps. However, more complicated geometry may require increasing the bitmap resolution in order to represent geometric details with desired accuracy. With 3D workspaces the situation is completely different, since we can no longer use the bitmap technique. Our experiments in 3D workspaces reported in [120] show that the higher cost of collision checking increases the duration of the learning phase. It affects less the query phase, since less collision checks are performed there. The results in [120] also show that the duration of the learning phase remains quite reasonable (on the order of minutes), but they were obtained with simple 3D geometry (for example, the robot links were line segments). For more complicated geometries, the use of an iterative collision checker, like the one in [166], will be advantageous. The collision checker in [166] considers successive approximations of the objects and its running time, on the average, does not depend much on the geometric complexity of the scenes. RPP is another planner that heavily relies on collision checking. For long we ran RPP on geometrically simple problems; but, recently, we used it to automatically animate graphic 3D scenes of complex geometry [124] using the above iterative collision checker. We observed no dramatic slowdown of the RPP planner.

A challenging goal would now be to extend the method to dynamic scenes. One first question is: how should a roadmap computed for a given workspace be updated if a few obstacles are removed or added? The work in [67, 85] discusses how to deal with changes in the environment in the context of the hybrid planner presented in [85]. We hope that similar techniques could apply to our planner. Being able to plan when obstacles move will be very useful because then we could apply our method to scenes subject to small incremental changes. Such changes occur in many manufacturing (e.g., assembly) cells; while most of the geometry of such a cell is permanent and stationary, a few objects (e.g., fixtures) are added or removed between any two consecutive manufacturing operations. Similar incremental changes also occur in automatic graphic animation. A second question is: how should the learning and query phase be modified if some obstacles are moving along known trajectories? An answer to this question might consist of applying our roadmap method in the configuration×time space of the robot [98]. The roadmap would then have to be built as a directed graph, since local paths between any two nodes must monotonically progress along the time axis, with possibly additional constraints on their slope and curvature to reflect bounds on the robot's velocity and acceleration.

Chapter 9

Experimental Integration of Planning in a Distributed Control System

This chapter is based on the paper "Experimental Integration of Planning in a Distributed Control System," by G. Pardo-Castellote, Tsai-Yen Li, Yoshihito Koga, Robert H. Cannon, Jr., Jean-Claude Latombe, and Stan Schneider, published in Proc. of International Symposium on Experimental Robotics, *Kyoto, Japan, October 1993*

9.1 Introduction

Automation and ease-of-operation are two goals of robotic systems. Ideally, one would specify a high-level task such as an assembly and have it executed automatically. To achieve these goals, sophisticated software modules such as planners, user interfaces, controllers etc. are being developed. However, the complexity of these modules and the fact that they are often developed at different times by different groups of people make system integration and testing very time consuming and often problem-specific.

In a joint effort, the Computer Science Robotics Laboratoryand the Aerospace Robotics Laboratoryat Stanford University have developed a flexible experimental test-bed to explore these issues. Our goal is to achieve task-level operation on a distributed robotic system in a dynamic environment.

The experimental demonstration is illustrated in Figure 9.1. Two 4-DOF arms manipulate parts in a dynamic environment containing both *static* and *moving* obstacles and parts. The parts are supplied by a conveyor. A vision system identifies and tracks the moving parts which are picked-up by the robot *while in motion.* Several efficient path planning modules are also implemented to find trajectories to deliver and assemble parts while avoiding the obstacles in the workspace. Due to their size and weight, some of the parts require cooperative manipulation and regrasping by the

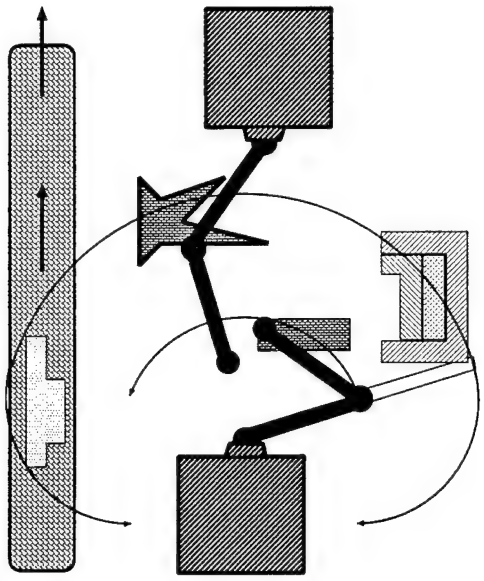


Figure 9.1: **Experimental Demonstration**

Experimental demonstration consisting of a robotic assembly in the presence of moving objects. The robot has two 4-DOF arms. The parts are delivered by a conveyor.

two arms whereas others are manipulated by a single arm. The user monitors and issues task-level commands using a graphical user interface.

9.2 System Architecture

Our experimental test-bed is composed of four modules as illustrated in Figure 9.2. The user interface receives state information from the robot and sensor systems and provides a graphical representation of the scene to the user. During operation, the high-level task is specified from the graphical user interface. The task planner/path planner receive continuous updates from the robot and sensors. The planners produce robot-commands primitives which are executed by the robot controller.

The simulator and the robot have the same interface to the other modules. This allows the simulator to masquerade as the robot for fast prototyping and testing of the rest of the system.

A novel *network-transparent, subscription-based* data-sharing scheme—the *Network Data Delivery Service* (NDDS)—facilitates communication among the different modules. It allows them to be distributed across different computer systems (with different processor architectures and operating

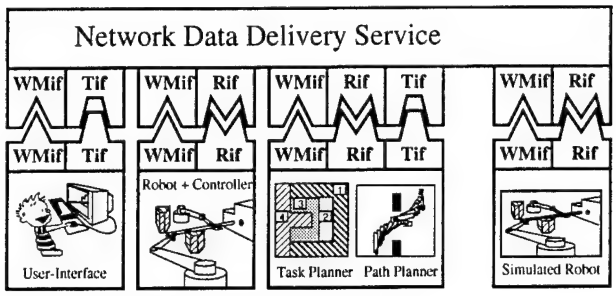


Figure 9.2: **System Architecture**

The overall system showing its four main modules. Each module communicates using one or more of the three interfaces: The World Model Interface (WMif), the Robot Interface (Rif) and the Task Interface (Tif). These modules are physically distributed. The Network Data Delivery Service plays the role of a bus providing the necessary interconnections.

systems[1]) and provides the necessary arbitration between data updates, enabling multiple users to operate and monitor the system concurrently.

The NDDS system [?] builds on the model of information producers (sources) and consumers (sinks). Producers register a set of data instances that they will produce, unaware of prospective consumers and "produce" the data at their own discretion. Consumers "subscribe" to updates of any data instances they require without concern for who is producing them. In this sense the NDDS is a "subscription-based" model. NDDS provides stateless (and hence robust) mechanisms to resolve multiple-producer conflicts and supports multiple-rate consumers.

The use of subscriptions to drastically reduces the update overhead over a classical client-server architecture. Occasional subscription requests, at low bandwidth, replace numerous high-bandwidth client requests. Latency is also reduced, as the outgoing request message time is eliminated.

NDDS differs from other distributed data-sharing schemes [61, 25, 11, 51] in its transparent support for multiple anonymous data-producers and consumers as well as in its fully-distributed, symmetrical implementation (which contains no privileged nodes).

All modules in the system communicate using one of the following three interfaces (built on top of NDDS): The *World Model* interface, the *Robot* interface and the *Task* interface. The functionality of two of these interfaces is summarized in tables 9.1 and 9.2.

This arrangement is analogous to a hardware bus as illustrated in Figure 9.2. NDDS plays the role of the physical interconnections. The three interfaces being similar to bus-access protocols. This approach is key to reducing system integration time and produces generic, reusable modules.

[1]The current demonstration involves DEC workstations, Sun Workstations and several VME-based real-time processors

9.3 Controller

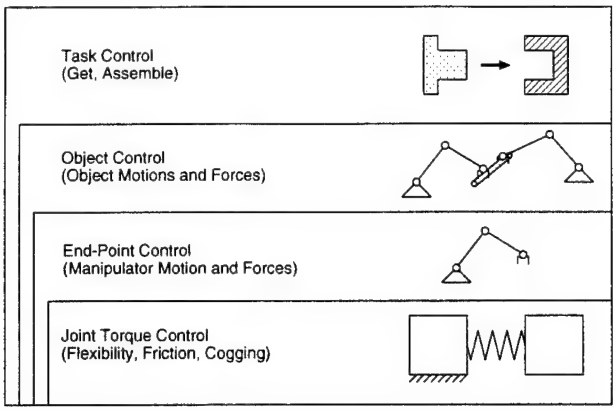


Figure 9.3: **Four-level control hierarchy for two-armed robot.**

We use a four layer hierarchy to control the two-armed robot. At the lower joint *level, we use joint-torque sensors to compensate for the non-idealities of the motor (cogging, non-linearity) and the joint dynamics induced by the joint flexibility. The next layer the* arm *level control can now assume ideal actuation (i.e. the motors deliver the desired torque to the link itself) and use a Computed Torque approach to compensate for the non-linear arm dynamics. The third* object *layer, is concerned with object behavior and assumes that the arms are virtual multi-dimensional actuators that apply torques to the object. The top layer implements elementary tasks such as object acquisition and release, insertions etc.*

The robot system consists of two four degree-of-freedom SCARA manipulators equipped with joint torque sensors, joint encoders and an end-point 6-DOF force sensor. An overhead vision system provides global sensing of the position of both the robot and objects. The robot is controlled from a VME-based real-time computer system.

The control software is organized in a four-layer hierarchy originally presented in [165]. The highest level of the control hierarchy illustrated in Figure 9.3 uses Finite State Machines (FSMs) to coordinate the actions of the two arms and react to both external and internal events. The current implementation uses three FSMs: A global FSM and individual ones for each of the two arms. The global FSM receives commands from the task-planner and may initiate a cooperative two-arm action and/or send stimuli to the individual arm FSMs. Capture of moving objects from the conveyor is achieved using FSM subchains. Subchains are FSM subprograms analogous to subroutines in conventional programming. For example, the subchain in the FSM illustrated in figure 9.6 is used to perform single-arm object acquisitions from a conveyor.

The next level of the hierarchy (object control) commands the arms to achieve the desired object behavior. The controller used enforces the Virtual Object Impedance Control policy [44]. The

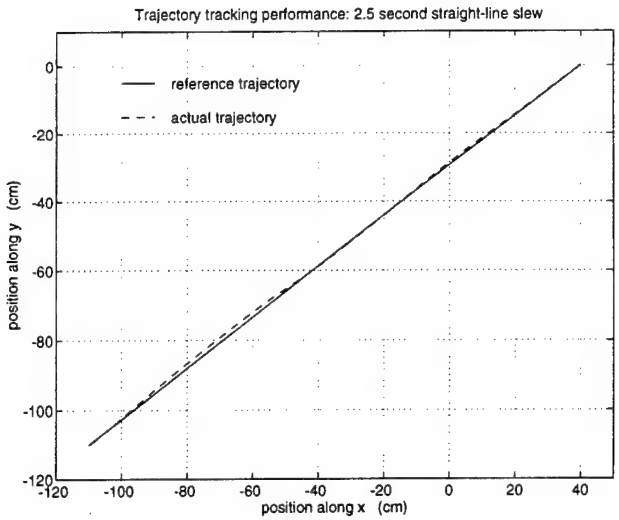


Figure 9.4: **Experimental tracking performance for right arm.**

Illustration of the tracking response of right arm. The reference is a fifth order polynomial trajectory for the arm endpoint commanding it to follow a 1.75 m straight line path in 2.5 sec. This trajectory requires accelerations of up to 4.3 m/s^2 (close to 1/2 g). The maximum tracking error is 1.4 cm.

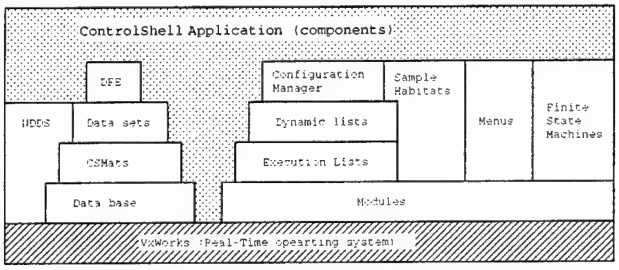


Figure 9.5: *ControlShell* **Structure.**

The right side of the diagram denotes the "execution" hierarchy; the left side is the "data" hierarchy. The application, consisting partially of a set of reusable components, has access to all facilities at every level. ControlShell *provides a layer on top of the real-time operating system VxWorks.*

third layer uses a Computed Torque approach to achieve dynamic arm control and, at the lowest (joint) level, higher bandwidth joint-torque control-loops are used to control the joint flexibility and compensate for the non-idealities of the motors. Figure 9.4 illustrates the trajectory-tracking performance of a single arm following a straight line.

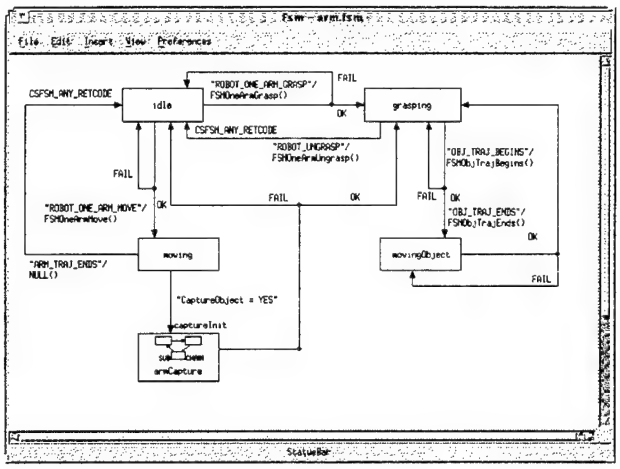


Figure 9.6: **Finite-State Machine (FSM) used to control an individual arm.**

Stimuli sent to the FSM (shown in quotes) cause the associated transition routines to execute. The return code of these transition routines determines the next state. Subchains are FSM "subprograms".

With the increased complexity demanded from distributed control-systems, the use of CASE tools that facilitate the programming task is becoming essential [10, 52, 48]. All the real-time software in this project has been implemented using the *ControlShell* framework. *ControlShell* [63] is a CASE tool that enables modular design and implementation of real-time software. It contains an object-oriented tool-set which provide a series of execution and data exchange mechanisms that capture both data and *temporal* dependencies. This allows a unique *component-based* approach to real-time software generation and management. *ControlShell* defines temporal events, and provides mechanisms for attaching routines to those events. It provides data structure specifications, and mechanisms for binding data and routines while resolving data dependencies. *ControlShell's* event-driven finite state machine provides the means to weave asynchronous events into a sequential execution stream. *ControlShell* also includes numerous code-generation and maintenance tools such as a graphical component editor, a finite state machine editor, a data-flow editor, an interactive menu facility etc. The structure of *ControlShell* is summarized in figure 9.5.

Object information from World Model	
location	object location in the global reference frame.
grasps	positions within the object where a grasp is possible
properties	Mass and inertia of a object, whether it can be moved by the robots etc.
shape	object shape for collision avoidance purposes.
Robot information from World Model	
location	robot location in the global reference frame.
joint values	value of each of the joint coordinates
joint limits	limits on the joint coordinates
Denavit-Hartenberg Parameters	Description of robot kinematics
state	Whether the robot is moving, grasping an object etc.

Table 9.1: Information Available using the World-Model interface

command	*meaning*
move object	Move an object that is being grasped. This command will provide a via-point collision-free path for the object. The robot is controlled using object impedance control.
move arms (operational space)	Move the arms. This command assumes the arms are not grasping an object. The command provides a via-point collision-free path for the arm-endpoints.
move arms (joint space)	Move the arms. This command assumes the arms are not grasping an object. The command provides a via-point collision-free path for the arms joints (This is provided to resolve kinematic ambiguities).
grasp	Grasp an object. This command specifies the object to be grasped.
un-grasp	Un-grasp an object.

Table 9.2: Robot-Command primitives available through the Robot interface.

9.4 Planner

The automatic generation of robot paths for accomplishing a user-specified task is one of the key elements in our automated robot system. Since our goal is to build such an interactive system, the on-line motion planning capability is crucial. Indeed, it is unacceptable for a user to specify a task and then have to wait a few minutes for the motion of the robots to be planned and executed. Due to the high computational cost of robot motion planning [6, 58], we focus our effort on developing effective strategies and efficient path planning algorithms to achieve on-line performance.

Alami, Siméon, and Laumond [39] have proposed and implemented an algorithm for the case of one robot and several movable objects. They assume both a finite number of possible grasps and a finite number of object placements within the workspace. The robot grasps one object at a time and the remaining (currently non-grasped) objects are treated as static obstacles. Their algorithm first considers all the possible arrangements of the robot grasping the object and then decomposes into cells the collision-free subset of these arrangements. These cells are subsequently connected by links which correspond to regrasping operations of the robot. The resulting graph, called the *Manipulation Graph*, is then searched for a path.

In [13], the authors consider a similar scenario dealing with two robots and multiple moving objects on a conveyor belt. The authors present an A^* algorithm to search for a locally-optimal motion sequence for a given assembly represented as a AND-OR graph. In this work, they assume that only the last links of the arms may collide with each other and that the parts are fed slowly enough that the robots can always pick up objects from locations that do not deviate too much from their nominal positions.

Our research differs from these related work in that we consider a more general problem with no prior knowledge about the speed, the feeding rate, the picking position, the picking arm, or the sequence of parts.

The planning system consists of two modules: the *Task Planner* and the *Path Planner*. The task planner is responsible for determining how to utilize the resources (in our case, two manipulator arms) to accomplish the user specified task. The result is a decomposition of the problem into subtasks, which are then sent to the path planner in order to find the necessary motion of the manipulators.

9.4.1 Task Planner

The role of the task planner is first determine how to solve the user-specified task (e.g. put object A at location P), then make use of the path planning algorithms to actually find the sequence of manipulator motions to complete the task and, finally to send the result to the robots in terms of the robot-command primitives listed in table 9.2.

The user-specified task may involve moving multiple objects to a set of specified goals. Because of this, the task-planner will break the task down into subtasks and, among other things, determine

which object to manipulate first. The criteria for choosing the next object to manipulate is priority-based. The priority of an object is based on the expected time that the (moving) object will take before it leaves the work cell. For example, static objects have lower priorities than objects on the conveyer belt. Once the next object to be manipulated is identified, a manipulator arm(s) is selected to pick-up and deliver the object. The criteria used to select the arm(s) takes into account whether the object/task requires two-arm manipulation, which arm(s) is free (i.e. not currently involved in another task), which arm is closer to the object, the expected time it will take for the object to leave the workspace of each arm, etc. At this point, rather than solving the complete manipulation path, that is moving the arm to grasp the object, grasping it, carrying it to the goal, and then ungrasping it, the task planner further divides the problem into two subtasks. The first subtask is moving the arm to grasp the object, while the second subtask is to deliver the object to its goal.

The task planner runs in a loop selecting the highest priority subtask to plan according to the current state of the world (this state includes both user-specified tasks and subtasks that are already "in-progress"). For example, if there are objects that need to be moved within the workcell, the task planner will detect this at the start of the loop, select the object with highest priority, and invoke the path planner to find a trajectory for one (or both) of the arms to grasp the object. Each time around the loop only the subtask with the highest priority is sent to the path planner. In the event that the planner fails to solve the highest priority subtask, the subtask with the next highest priority is attempted. Once a plan is found, it is broken down into individual robot-command primitives and sent to the robot for execution. Once the motion of the robot (as a response to the command) is detected, the subtask becomes "in-progress" and the task planner returns to the top of the loop. In the event that the execution of this subtask become questionable (e.g. the obstacles have moved or the goal position has been changed) the task planner will first determine if the plan is still valid and if not it will replan accordingly.

9.4.2 Path Planner

The subtasks that are requested by the task planner to be solved are the following. Note that a free arm refers to a robot arm that is not committed to any subtask.

- Move one arm to grasp a static object while the other arm is free.
- Move one arm to grasp a static object while the other arm is moving.
- Move one arm to catch a moving object while the other arm is free.
- Move one arm to catch a moving object while the other arm is moving.
- Deliver the object that is grasped to its goal location while the other arm is free.
- Deliver the object that is grasped to its goal location while the other arm is holding another object.

- Move two arms to grasp a static object.
- Move two arms to catch a moving object.

Notice that we do not consider the case where both arms deliver two independent objects at the same time or the case where one arm is moving and the other arm delivers its object to the goal. For these cases we decouple the problem and plan the motion of the arms sequentially. Our reasoning is that though the *robot execution time* for this decoupled approach may be slightly longer than if the arms moved simultaneously, the *planning time* to find the sequential motion will be significantly shorter than the time to find the simultaneous motion of both arms. Indeed since efficiency is the key issue in on-line motion planning, we make several assumptions to simplify the path planning problem associated to each subtask and build a library of efficient primitives to solve them.

In our scenario, reasonable simplifications can be made to reduce the size of the search spaces implied by each subtask, and thus reducing the time required to solve them. Due to the fact that the first two links of the SCARA-type arms move in a plane and our assembly task only involves pick-and-place operations, we simplify the motion planning problem in the three dimensional workspace into a problem of two dimensions. This assumes that when the end-effectors of the arms are as high up as they can go, the arms can move in an unrestricted manner above the obstacles in the workspace. This reduces the search space of each arm from three dimensions to two[2]. Once the arm(s) are grasping an object, the size of the object is considered to be such that it can collide with the obstacles in the workspace–that is, the arms are unable to lift the object above the obstacles. The result is a three dimensional search space which fortunately still presents little difficulty for fast computation.

For each of the aforementioned search spaces, we build a path planning primitive to find a collision-free path connecting two configurations in the particular search space. These primitives are extremely efficient and can find paths in a fraction of a second. Some of them are based on existing algorithms [130, 19], while the others are completely new. We then associate a strategy with each subtask which utilizes these primitives to solve it. For example, the subtask of moving one arm to catch a moving object while the other arm is free has the following strategy. First the path of the moving arm to grasp the object is found while ignoring the presence of the free arm. The second step is to have the free arm comply with the motion of this moving arm. If the subtask is solved, the corresponding motion of the robots is returned to the task planner. Otherwise, the task planner is notified that the particular subtask could not be solved.

Our experiments demonstrate that this planning approach yields on-line performance.

9.5 Discussion and Conclusions

This paper has presented the overall design of an experimental test bed developed to explore the integration of planning in distributed control systems.

[2] the complexity of motion planning grows exponentially with the dimension of the search space

We have concentrated on several key aspects of this problem. First, we have developed a powerful distributed network communication architecture (NDDS), and implemented it within a generic real-time programming framework (ControlShell). We have defined clear interface specifications, and implemented simple-yet-powerful robotic system modules within that architecture.

In particular, we have developed and implemented a graphical user interface, a four-level dynamic control hierarchy, an on-line simulator, and an extremely fast on-line motion and task planning capability. The unique aspects of these modules are described briefly above.

This paper describes work in progress by the authors. Our experiments with *non-moving* objects have shown the power of our approach. Numerous tests in *simulation* have shown the system to be able to operate in the presence of moving objects. We will experimentally demonstrate assembly with moving parts in the immediate future.

Bibliography

[1] A. Hörmann and U. Rembold. Development of an Advanced Robot for Autonomous Assembly. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2452–57, Sacramento, CA, May 1991.

[2] J. S. Albus, R. Lumia, and H. McCain. Hierarchical control of intelligent machines applied to space station telerobots. 24(5):535 – 541, September 1988. Three (distinct, but inter-related) hierarchies: task decomposition, world model, and sensory processing.

[3] J. S. Albus, H. G. McCain, and R. Lumia. NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM). Technical Report 1235, NIST, April 1989.

[4] Lawrence Alder. *Control of a Flexible-Link Robotic Arm Manipulating An Unknown Dynamic Payload.* PhD thesis, Stanford University, Stanford, CA 94305, February 1993. Also published as SUDAAR 632.

[5] Henri E. Bal, Jenniffer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computer Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[6] J.F. Canny. *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1988.

[7] Vincent W. Chen. *Experiments in Adaptive Control of Multiple Cooperating Manipulators on a Free-Flying Space Robot.* PhD thesis, Stanford University, Stanford, CA 94305, December 1992. Also published as SUDAAR 631.

[8] R. S. Chin and S. T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–123, March 1991.

[9] Douglas E. Comer. *Internetworkning with TCP/IP*, volume 1-3. Prentice Hall, Englewood Cliffs, N.J., 2 edition, 1991-92.

[10] A. Joubert D. Simon. The orccad: Towards an open robot controller computer aided design system. Research Report 1396, INRIA, February 1991.

[11] Chris Fedor. TCX task communications. School of computer science / robotics institute report, Carnegie Mellon University, 1993.

[12] S. Fleury, M. Herrb, and R. Chatila. Design of a Modular Architecture for Autonomous Robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3508–13, San Diego, CA, May 1994.

[13] Li-Chen Fu and Yung-Jen Hsu. Fully automated two-robot flexible assembly cell. In *IEEE International Conference on Robotics and Automation*, pages 332–338, Atlanta, Georgia, USA, May 1993.

[14] G. Hirzinger and J Dietrich. A computer architecture for intelligent machines. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Nice, France, May 1992.

[15] Andreas Hormann. On-Line Planning of Action Sequences for a Two-Arm Manipulator System. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1109–1114, Nice, France, May 1992.

[16] Real-Time Innovations Inc. *ControlShell: Object Oriented Framework for Real-Time Software User's Manual.* 954 Aster, Sunnyvale, California 94086, 4.2 edition, August 1993.

[17] Integrated Systems, Inc., 2500 Mission College Boulevard, Santa Clara, CA 95054. *ISI Product Literature*, 1990-93.

[18] J. P. Jones, P. L. Butler, S. E. Johnston, and T. G. Heywood. Hetero Helix: synchronous and asynchronous control systems in heterogeneous distributed networks. *Robotics and Autonomous Systems*, 10(2-3):85–99, 1992.

[19] Y. Koga and J.C. Latombe. Experiments in dual-arm manipulation planning. In *IEEE International Conference on Robotics and Automation*, pages 2238–2245, Nice, France, May 1992.

[20] Ross Koningstein. *Experiments in Cooperative-Arm Object Manipulation with a Two-Armed Free-Flying Robot.* PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, October 1990. Also published as SUDAAR 597.

[21] J.C. Latombe. *Robot Motion Planning.* Kluger Academic Publishers, Boston, MA, 1991.

[22] Tsai-Yen Li and Jean-Claude Latombe. On-Line Motion Planning for Two Robot Arms in a Dynamic Environment. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, Nagoya, Japan, May 1995.

[23] R. Lumia, J. L. Michaloski, R. Russel, T. E. Wheatley, P. G. Backes, S. Lee, and R. D. Steele. Unified Telerobotic Architecture Project (UTAP) Interface Document. Technical report, NIST, Intelligent Systems Division, NIST, Gaithersburg, MD, June 18 1994.

[24] The MathWorks, Inc., Cochituate Place, 24 Prime Park Way, Natick MA 01760. *Product Literature*, 1990-93.

[25] MBARI (Monterrey Bay Aquarium Research Institute. Data manager user's guide. Internal Documentation, 1991.

[26] Sun Microsystems. XDR: External Data Representation Standard. Internet Network Working Group Requets for Comments RFC 1014, Network Information Center, SRI International, June 198t.

[27] D.J. Miller and R.C. Lennox. An Object-Oriented Environment for Robot System Architectures. *IEEE Control Systems Magazine*, 11(2):14–23, February 1991.

[28] S. Narasimhan, D. Siegel, and J. M. Hollerbach. Condor: A revised architecture for controlling the Utah/MIT hand. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 446–449, Philadelphia, PA, April 1988.

[29] F. R. Noreils. Toward a robot architecture integrating cooperation between mobile robots. *The International Journal of Robotics Research*, 12(1):79–98, February 1993.

[30] Celia M. Oakley. *Experiments in Modelling and End-Point Control of Two-Link Flexible Manipulators.* PhD thesis, Stanford University, Department of Mechanical Engineering, Stanford, CA 94305, April 1991.

[31] G. Pardo-Castellote and S. A. Schneider. The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications. In *Proceedings of the International Conference on Robotics and Automation*, San Diego, CA, May 1994. IEEE, IEEE Computer Society.

[32] Gerardo Pardo-Castellote. *Experimental Integration of Planning and Control for a Intelligent Manufacturing Workcell.* PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, CA 94305, June 1995.

[33] Gerardo Pardo-Castellote, Tsai-Yen Li, Yoshihito Koga, Robert H. Cannon Jr., Jean-Claude Latombe, and Stan Schneider. Experimental Integration of Planning in a Distributed Control System. In Tsuneo Yoshikawa and Fumio Miyazaki, editors, *Experimental Robotics III: The third International Symposium*, volume 200 of *Lecture Notes in Control and Information Sciences*, pages 50–61. Springer-Verlag, Kyoto Japan, October 28-30 1993.

[34] D. W. Payton, J. K. Rosenblatt, and D. M. Keirsey. Plan Guided Reaction. *IEEE Transactions on Systems, Man and Cybernetics*, 20(6):1370–82, Nov-Dec 1990.

[35] Lawrence E. Pfeffer. *The Design and Control of a Two-Armed, Cooperating, Flexible-Drivetrain Robot System.* PhD thesis, Stanford University, Stanford, CA 94305, December 1993. Also published as SUDAAR 644.

[36] Judson P. Jones Philip L. Butler. A Modular Control Architecture for Real-Time Synchronous and Asynchronous Systems. In *Proceedings of the SPIE - Applications of Artificial Intelligence: Machine Vision and Robotics*, volume 1964, pages 287–298. SPIE, 1993.

[37] J. Postel. User datagram protocol. RFC 768, June 1980.

[38] R. Quintero and A.J. Barbera. A Real-Time Control System Methodology for Developing Intelligent Control Systems. Technical Report NISTIR 4936, NIST, October 1992.

[39] T. Siméon R. Alami and J.P. Laumond. A geometrical approach to planning manipulation tasks: The case of discrete placements and grasps. In H. Miura and S. Arimoto, editors, *Robotics Research 5*, pages 453–459, Cambridge, MA, 1990. The MIT Press.

[40] Ready Systems, Inc. *VRTX User's Manual*, 1993.

[41] Real-Time Innovations, Inc., 954 Aster, Sunnyvale, CA 94086. *NDDS: The Network Data-Delivery Service User's Manual*, 1.7 edition, November 1994.

[42] Eberhardt Rechtin. *Systems Architecting: Creating and Building Complex Systems.* Prentice-Hall, first edition, 1991.

[43] S. Schneider. *Experiments in the Dynamic and Strategic Control of Cooperating Manipulators.* PhD thesis, Stanford University, Stanford, CA 94305, September 1989. Also published as SUDAAR 586.

[44] S. Schneider and R. H. Cannon. Object Impedance Control for Cooperative Manipulation: Theory and Experimental Results. *IEEE Transactions on Robotics and Automation*, 8(3), June 1992. Paper number B90145.

[45] S. A. Schneider and R. H. Cannon. Experimental Object-Level Strategic Control With Cooperating Manipulators. *The International Journal of Robotics Research*, 12(4):338–350, August 1993.

[46] S. A. Schneider, V. Chen, and G. Pardo-Castellote. Object-Oriented Framework for Real-Time System Development. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Nagoya, Japan, May 1995. IEEE, IEEE Computer Society.

[47] S. A. Schneider, V. W. Chen, and G. Pardo-Castellote. ControlShell: A Real-Time Software Framework. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space*, volume II, pages 870–7, Houston, TX, March 1994. AIAA, AIAA.

[48] Reid Simmons and Chris Fedor. Task control architecture programmer's guide. School of computer science / robotics institute report, Carnegie Mellon University, 1992.

[49] Simon. The orccad system. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Nice, France, May 1992.

[50] Software Components Group, Inc., 4655 Old Ironsides Drive, Santa Clara, CA 95054. *pSOS+/68K Real-Time, Multi-processing Operating System Kernel User's Manual*, 0.4.a edition, January 1989.

[51] Sparta, Inc., 7926 Jones Branch Drive, McLean, VA 22102. *ARTSE product literature.*

[52] D. B. Stewart, D. E. Schmitz, and P.K. Khosla. Chimera ii: A real-time multiprocessing environment for sensor-based robot control. In *Proceedings of the IEEE International Symposium on Intelligent Control*, Albany, NY, September 1989.

[53] S. W. Tilley, C. M. Francis, K. Emerick, and M. G. Hollars. Preliminary results on noncolocated torque control of space robot actuators. In *Proceedings of the NASA Conference on Space Telerobotics*, Pasadena, CA, February 1989. NASA.

[54] S. W. Tilley, M. G. Hollars, and K. S. Emerick. Experimental control results in a compact space robot actuator. In *Proceedings of the ASME Winter Annual Meeting*, San Francisco, CA, December 1989.

[55] Christopher R. Uhlik. *Experiments in High-Performance Nonlinear and Adaptive Control of a Two-Link, Flexible-Drive-Train Manipulator.* PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, CA 94305, May 1990. Also published as SUDAAR 592.

[56] M. A. Ullman. *Experiments in Autonomous Navigation and Control of Multi-Manipulator Free-Flying Space Robots.* PhD thesis, Stanford University, Stanford, CA 94305, March 1993. Also published as SUDAAR 630.

[57] H. H. Wang, R. L. Marks, S. M. Rock, and M. J. Lee. Task-Based Control Architecture for an Untethered, Unmanned Submersible. In *Proceedings of the 8th Annual Symposium of Unmanned Untethered Submersible Technology*, pages 137–147. Marine Systems Engineering Laboratory, Northeastern University, September 1993.

[58] G. Wilfong. Motion planning in the presence of movable obstacles. In *4th ACM Symp. of Computational Geometry*, pages 279–288, Urbana-Champaign, Illinois, June 1988.

[59] T. Williams. Fiber network supports distributed real-time systems. *Computer Design*, 29(17):60–62, September 1990. dd.

[60] Wind River Systems, Inc., 1351 Ocean Ave., Emeryville, CA 94608. *VxWorks User's Manual*, 1988-1993.

[61] J. D. Wise and Larry Ciscon. *TelRIP Distributed Applications Environment Operating Manual.* Rice University, Houston Texas, 1992. Technical Report 9103.

[62] F. Zanichelli, S. Caselli, A. Natali, and A. Omicini. A Multi-Agent Framework and Programming Environment for Autonomous Robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3501–7, San Diego, CA, May 1994.

[63] —— 1993. *ControlShell: Object Oriented Framework for Real-Time Software – User's Manual.* Sunnyvale: Real-Time Innovations Inc., 4.2 Edition.

[64] J. M. Ahuactzin, E.-G. Talbi, P. Bessière, and E. Mazer. Using genetic algorithms for robot motion planning. *10th Europ. Conf. Artific. Intelligence.* 671–675. John Wiley and Sons, Ltd., London, England, 1992.

[65] Alami, R., Siméon, T., and Laumond, J.P. 1990. A Geometrical Approach to Planning Manipulation Tasks: The Case of Discrete Placements and Grasps. *Robotics Research 5*, ed. H. Miura and S. Arimoto. Cambridge: MIT Press, pp. 453-459.

[66] Ayache, N., *Artificial Vision for Mobile Robots: Stereo Vision and Multisensory Perception*, The MIT Press, Cambridge, MA, 1991.

[67] M. Barbehenn, P.C. Chen, and S. Hutchinson. An efficient hybrid planner in changing environments. *Proc. IEEE Int. Conf. Robotics and Automation*, 2755–2760, San Diego, CA, May 1994.

[68] Barraquand, J. and Latombe, J.C. 1991. Robot Motion Planning: A Distributed Representation Approach. *Int. J. Robotics Res.* 10(6):628-649.

[69] Barraquand, J., Langlois, B., and Latombe, J.C. 1992. Numerical Potential Field Techniques for Robot Path Planning. *IEEE Tr. Systems, Man, and Cybernetics.* 22(2):224–241.

[70] J. Barraquand and P. Ferbach. Path planning through variational dynamic programming. *Proc. IEEE Int. Conf. Robotics and Automation*, 1839–1846, San Diego, CA, May 1994.

[71] B. Barraquand, L.E. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, and P. Raghavan. A random sampling scheme for robot path planning. *Robotics Research*, G. Giralt and G. Hirzinger (ed.), North Holland, 1996, to appear. (Proc. 7th Int. Symp. on Robotics Research, Herrsching, Germany, October 1995.)

[72] S. Berchtold and B. Glavina. A scalable optimizer for automatically generated manipulator motions. *Proc. IEEE/RSJ/GI Int. Conf. Intelligent Robots and Systems*, 1796-1802, München, Germany, 1994.

[73] Brafman, R.I., Latombe, J.C., and Shoham, Y., Towards Knowledge-Level Analysis of Motion Planning, *Proc. of the 11th Nat. Conf. on Artificial Intelligence*, AAAI, Washington, DC, July 1993.

[74] Branicky, M.S. and Newman, W.S. 1990. Rapid Computation of Configuration Space Obstacles. *Proc. 1990 IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 304-310.

[75] Briggs, A.J., An Efficient Algorithm for One-Step Planar Compliant Motion Planning with Uncertainty, *Proc. of the 5th Annual Symp. on Comp. Geom.*, Saarbruchen, Germany, 1989.

[76] Brooks, R.A., Symbolic Error Analysis and Robot Planning, *Int. J. of Robotics Research*, 1(4):29-68, 1982.

[77] Buckley, S.J., *Planning and Teaching Compliant Motion Strategies*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 1986.

[78] Canny, J.F. and Reif, J.H., New Lower-Bound Techniques for Robot Motion Planning Problems, *Proc. of the 28th SYMP. on the FOCS*, Los Angeles, 49-60, 1987.

[79] J.F. Canny. *The complexity of robot motion planning.* MIT Press, Cambridge, MA, 1988.

[80] Canny, J.F., On Computability of Fine Motion Plans, *Proc. of the IEEE Int. Conf.on Robotics and Automation* Scottsdale, AZ, 177-182, 1989.

[81] J.F. Canny and M.C. Lin. An opportunistic global path planner. *Proc. IEEE Int. Conf. Robotics and Automation*, 1554–1559, Cincinnati, OH, 1990.

[82] D. Chalou and M. Gini. Parallel robot motion planning. *Proc. of IEEE Int. Conf. Robotics and Automation*, 24–51, Atlanta, GA, 1993.

[83] Chang, H.S. and Li, T.Y. 1995. Assembly Maintainability Study with Motion Planning. *Proc. 1995 IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 709-714.

[84] Chapman, D. and Agre, P.E. Abstract Reasoning as Emergent from Concrete Activity, *Reasoning about Actions and Plans*, edited by Georgeff, M.P. and Lansky, A.L, Morgan Kaufmann Publishers, Los Altos, CA, 1986, pp. 411-424.

[85] P.C. Chen. Improving path planning with learning. *Proc. Machine Learning Conference*, 55–61, 1992.

[86] P.C. Chen. *Adaptive path planning in changing environments.* Report SAND92-2744, Sandia National Laboratories, 1993.

[87] P.C. Chen and Y.K. Hwang. SANDROS: A motion planner with performance proportional to task difficulty. *Proc. of IEEE Int. Conf. Robotics and Automation,* 2346-2353, Nice, France, 1992.

[88] H. Choset and J. Burdick. Sensor based planning and nonsmooth analysis. *Proc. of IEEE Int. Conf. Robotics and Automation*, 3034-3041, San Diego, CA, 1994.

[89] Craig, J.J. 1986. *Introduction to Robotics. Mechanics and Control.* Reading:Addison-Wesley.

[90] Crowley, J.L., World Modeling and Position Estimation for a Mobile Robot Using Ultrasonic Ranging, *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Scottsdale, AZ, 674-680, 1989.

[91] Dakin, G.A. and Popplestone, R.J., Augmenting a Nominal Assembly Motion Plan with a Compliant Behavior. *Proc. of the 9th Nat. Conf. on Artificial Intelligence*, Anaheim, CA, 653-658, July 1991.

[92] Donald, B.R., A Geometric Approach to Error Detection and Recovery for Robot Motion Planning with Uncertainty, *Artificial Intelligence J.*, 52(1-3):223-271, 1988.

[93] Donald, B.R., The Complexity of Planar Compliant Motion Planning Under Uncertainty, *Algorithmica*, 5:353-382, 1990.

[94] Donald, B.R. and Jennings, J., Constructive Recognizability for Task-Directed Robot Programming, *J. of Robotics and Autonomous Systems*, 9, 41-74, 1992.

[95] Drummond, M. Situated Control Rules, *Proc. of the First Int. Conf. on Principles of Knowledge Representation and Reasoning*, Morhan Kaufmann Publishers, Los Altos, CA, 1989, pp. 103-113.

[96] Dufay, B. and Latombe, J.C., An Approach to Automatic Robot Programming Based on Inductive Learning, *Int. J. of Robotics Research*, 3(4):3-20, 1984.

[97] Erdmann, M., *On Motion Planning with Uncertainty*, Tech. Rep. 810, AI Lab., MIT, Cambridge, MA, 1984.

[98] Erdmann, M. and Lozano-Pérez, T. 1987. On Multiple Moving Objects. *Algorithmica.* 2(4):477-521.

[99] Erdmann, M., Randomization in Robot Tasks, *Int. J. of Robotics Research*, 11(5):399-436, 1992.

[100] Erdmann, M., *Towards Task-Level Planning: Action-Based Sensor Design*, Tech. Rep. CMU-CS-92-116, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, February 1992.

[101] B. Faverjon and P. Tournassoud. A local approach for path planning of manipulators with a high number of degrees of freedom. *Proc. IEEE Int. Conf. Robotics and Automation*, 1152–1159, Raleigh, NC, 1987.

[102] Faverjon, B. and Tournassoud, P. 1990. A Practical Approach to Motion-Planning for Manipulators with Many Degrees of Freedom. *Robotics Research 5*, ed. H. Miura and S. Arimoto. Cambridge:MIT Press, pp. 425-433.

[103] Fox, A. and Hutchinson, S., *Exploiting Visual Constraints in the Synthesis of Uncertainty-Tolerant Motion Plans*, Tech. Rep. UIUC-BI-AI-RCV-92-05, The University of Illinois at Urbana-Champaign, October 1992.

[104] Friedman, J., *Computational Aspects of Compliant Motion Planning*, Ph.D. Dissertation, Report No. STAN-CS-91-1368, Dept. of Computer Science, Stanford University, Stanford, CA, 1991.

[105] Fujimura, K. 1991. *Motion Planning in Dynamic Environments.* New York:Springer-Verlag.

[106] Fujimura, K. 1994. Motion Planning Amid Transient Obstacles. *Int. J. Robotics Res.*. 13(5):395-407.

[107] G.-J. Giezeman. *PlaGeo—A library for planar geometry.* Tech. Rep., Dept. Comput. Sci., Utrecht Univ., Utrecht, The Netherlands, 1993.

[108] Graux, L., Millies, P., Kociemba, P.L., and Langlois, B. 1992. Integration of a Path Generation Algorithm into Off-Line Programming of AIRBUS Panels. *Aerospace Automated Fastening Conf. and Exp.*, SAE Tech. Paper 922404.

[109] Guibas, L., Ramshaw, L., and Stolfi, J. 1983. A Kinetic Framework for Computational Geometry. *Proc. FOCS.* New York:IEEE, pp. 100-111.

[110] K. Gupta and Z. Gou. Sequential search with backtracking. *Proc. of IEEE Int. Conf. Robotics and Automation*, 2328–2333, Nice, France, 1992.

[111] Gupta, K.K. and Zhu, X. 1994. Practical Motion Planning for Many Degrees of Freedom: A Novel Approach Within Sequential Framework. *Proc. IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 2038-2043.

[112] Gupta, K.K. and Zucker, S.W. 1986. Toward Efficient Trajectory Planning: Path Velocity Decomposition. *Int. J. of Robotics Res.*. 5:72-89.

[113] Th. Horsch, F. Schwarz, and H. Tolle. Motion planning for many degrees of freedom - random reflections at C-space obstacles. *Proc. IEEE Int. Conf. Robotics and Automation*, 3318–3323, San Diego, CA, 1994.

[114] Hutchinson, S., "Exploiting Visual Constraints in Robot Motion Planning," *Proc. of the IEEE Int. Conf. of Robotics and Automation*, Sacramento, CA, 1722-1727, 1991.

[115] Gottschlich, S.N. and Kak, A.C., Dealing with Uncertainty in CAD-Based Assembly Motion Planning, *Proc. of the 9th Nat. Conf. on Artificial Intelligence*, Anaheim, CA, 646-652, July 1991.

[116] L.E. Kavraki. Computation of configuration-space obstacles using the fast fourier transform. *Proc. IEEE Int. Conf. Robotics and Automation*, 255–261, Atlanta, GA, 1993. To appear in *IEEE Tr. Robotics and Automation.*

[117] L.E. Kavraki. *Random networks in configuration space for fast path planning*, Ph.D. Thesis, Tech. Rep. STAN-CS-95-1535, Dept. Comput. Sci., Stanford Univ., Stanford, CA, January 1995.

[118] L.E. Kavraki and J.-C. Latombe. *Randomized preprocessing of configuration space for fast path planning.* Tech. Rep. STAN-CS-93-1490, Dept. Comput. Sci., Stanford Univ., Stanford, CA, September 1993.

[119] Kavraki, L. and Latombe, J.C. 1994. Randomized Preprocessing of Configuration Space for Fast Path Planning. *Proc. IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 2138-2145.

[120] L.E. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for path planning: Articulated robots. *Proc. IEEE/RSJ/GI Int. Conf. Intelligent Robots and Systems*, 1764-1772, Germany, 1994.

[121] L.E. Kavraki, J.-C. Latombe, R. Motwani, and P. Raghavan. Randomized query processing in robot path planning. *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, 353-362, Las Vegas, NV, May 1995,

[122] O. Khatib, "Object Manipulation in a Multi-Effector Robot System," *Robotics Research 4*, R. Bolles and B. Roth, eds., MIT Press, Cambridge, MA, 1988, pp. 137-144.

[123] D.E. Koditschek. Exact robot navigation by means of potential functions: some topological considerations. *Proc. IEEE Int. Conf. Robotics and Automation*, 1-6, Raleigh, NC, 1987.

[124] Koga, Y., Kondo, K., Kuffner, J., and Latombe, J.C., 1994. Planning Motions with Intentions. *Proc. SIGGRAPH'94*. New York:ACM, pp. 395-408.

[125] Koga, Y., Lastennet, T., Li, T.Y., and Latombe, J.C. 1992 (Tokyo). Multi-Arm Manipulation Planning. *Proc. 9th Int. Symp. on Automation and Robotics in Construction.* pp. 281-288.

[126] Koga, Y. and Latombe, J.C. 1992. Experiments in Dual-Arm Manipulation Planning. *Proc. IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 2238-2245.

[127] Koga, Y. and Latombe, J.C. 1994. On Multi-Arm Manipulation Planning. *Proc. IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 945-952.

[128] K. Kondo. Motion planning with six degrees of freedom by multistrategic bidirectional heuristic free-space enumeration. *IEEE Tr. on Robotics and Automation*, 7(3):267–277, 1991.

[129] K. Kondo, *Inverse Kinematics of a Human Arm*, in preparation, 1993.

[130] Latombe, J.C. 1991. *Robot Motion Planning.* Boston:Kluwer.

[131] Latombe, J.C. 1991. A Fast Path Planner for a Car-Like Indoor Mobile Robot. *Proc. 9th Nat. Conf. Artificial Intelligence.* Menlo Park:AAAI, pp. 659-665.

[132] Latombe, J.C., Lazanas, A., and Shekhar, S., Robot Motion Planning with Uncertainty in Control and Sensing, *Artificial Intelligence J.* 52(1):1-47, 1991.

[133] Laugier, C. and Théveneau, P., Planning Sensor-Based Motions for Part-Mating Using Geometric Reasoning Techniques, *Proc. of European Conf. on Artificial Intelligence*, Brighton, UK, 1986.

[134] J.P. Laumond and R. Alami, *A Geometrical Approach to Planning Manipulation Tasks: The Case of a Circular Robot and a Movable Circular Object Amidst Polygonal Obstacles*, Rep. No. 88314, LAAS, Toulouse, 1989.

[135] J.P. Laumond and R. Alami, *A Geometrical Approach to Planning Manipulation Tasks in Robotics*, Rep. No. 89261, LAAS, Toulouse, 1989.

[136] J.P. Laumond, M. Taix, and P. Jacobs. A motion planner for car-like robots based on a global/local approach. Proc. IEEE Internat. Workshop Intell. Robot Syst. 765–773. 1990.

[137] Lazanas, A. and Latombe, J.C., Landmark-Based Robot Navigation, *Proc. of the 10th Nat. Conf. on Artificial Intelligence*, San Jose, 697-702, July 1992.

[138] Lazanas, A. and Latombe, J.C., *Landmark-Based Robot Navigation*, Tech. Rep. STAN-CS-92-1428, Dept. of Computer Science, Stanford Univ., Stanford, CA, 1992 (to appear in *Algorithmica*).

[139] Lazanas, A. and Latombe, J.C., *Motion Planning with Uncertainty: A Landmark Approach*, Tech. Rep., Dept. of Computer Science, Stanford Univ., Stanford, CA, 1993.

[140] Lengyel, J., Reichert, M., Donald, B.R. and Greenberg, P. 1990. Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. *Proc. SIGGRAPH'90.* New-York:ACM, pp. 327-336.

[141] Leonard, J.J. and Durrant-Whyte, H.F., Mobile Robot Localization by Tracking Geometric Beacons, *IEEE Tr. on Robotics and Automation*, 7(3), 376-382, 1991.

[142] Levitt, T.S., Lawton, D.T., Chelberg, D.M. and Nelson, P.C., Qualitative Navigation, *Proc. of DARPA Image Understanding Workshop*, Los Angeles, CA, 447-465, 1987.

[143] Lozano-Pérez, T.,*The Design of a Mechanical Assembly System*, Technical Report AI-TR 397, Artificial Intelligence Laboratory, MIT, 1976.

[144] Lozano-Pérez, T. 1983. Spatial Planning: A Configuration Space Approach. *IEEE Tr. on Computers.* 32(2):108-120.

[145] Lozano-Pérez, T., Mason, M.T., and Taylor, R.H., Automatic Synthesis of Fine-Motion Strategies for Robots, *Int. J. of Robotics Research*, 3(1):3-24, 1984.

[146] T. Lozano-Pérez et al., "Handey: A Task-Level Robot System," *Robotics Research 4*, R. Bolles and B. Roth, eds., MIT Press, Cambridge, MA, 1988, pp. 29-36.

[147] Lozano-Pérez, T. 1991. Parallel Robot Motion Planning. *Proc. IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 1000-1007.

[148] T. Lozano-Pérez. Spatial planning: a configuration space approach. *IEEE Tr. on Computers*, 32:108–120, 1983.

[149] T. Lozano-Pérez and P. O'Donnel. Parallel robot motion planning. *Proc. IEEE Int. Conf. Rob. and Automation*, 1000–1007, Sacramento, CA, 1991.

[150] T. Lozano-Pérez and M.A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. of the ACM*, 22(10):560-570, 1979.

[151] Maciejewski, A.A. and Fox, J.J. 1993. Path Planning and the Topology of Configuration Space. *IEEE Tr. on Robotics and Automation.* 9(4):444-456.

[152] Mahadevan, S. and Connell, J., *Automatic Programming of Behavior-based Robots using Reinforcement Learning*, Research Rep., IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990.

[153] Mason, M.T., Automatic Planning of Fine Motions: Correctness and Completeness, *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Atlanta, GA, 492-503, 1984.

[154] J. Mastwijk. *Motion planning using potential field methods.* Master Thesis, Dept. Comput. Sci., Utrecht Univ., Utrecht, The Netherlands, August 1992.

[155] Natarajan, B.K., The Complexity of Fine Motion Planning, *Int. J. of Robotics Research*, 7(2):36-42, 1988.

[156] C. Ó'Dúnlaing and C.K. Yap. A retraction method for planning the motion of a disc. *J. of Algorithms*, 6:104-111, 1982.

[157] M. Overmars. *A random approach to motion planning.* Tech. Rep. RUU-CS-92-32, Dept. Comput. Sci., Utrecht Univ., Utrecht, The Netherlands, October 1992.

[158] M. Overmars and P. Švestka. A probabilistic learning approach to motion planning. In *Algorithmic Foundations of Robotics*, K. Goldberg et al. (eds.), A K Peters, Wellesley, MA, 19-37, 1995.

[159] Papadimitriou, C.H. and Yanakakis, M. 1991. Shortest Paths Without a Map. *Theoretical Computer Science.* 84:127-150.

[160] Pardo-Castellote, G. 1994. Experimental Integration of Planning and Control in a Distribited Robotic System. Ph.D. thesis, Stanford University, Department of Electrical Engineering.

[161] Pardo-Castellote, G., Li, T.Y., Koga, Y., Cannon, R.H., Latombe, J.C., and Schneider, S.A. 1993 (October 1993, Kyoto). Experimental Integration of Planning in a Distributed Control System. *Proc. Int. Symp. on Experimental Robotics.* pp. 217-222.

[162] Pardo-Castellote, G. and Schneider, S.A. 1994. The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications. *Proc. IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 2870-2876.

[163] J. Pertin-Troccaz, "Grasping: A State of the Art," in *The Robotics Review 1*, O. Khatib, J.J. Craig, and T. Lozano-Pérez, eds., MIT Press, Cambridge, MA, 1989, pp. 71-98.

[164] Pertin-Troccaz, J. and Puget, P., Dealing with Uncertainty in Robot Planning Using Program Proving Techniques, *Proc. of the 4th Int. Symp. on Robotics Research*, Santa-Cruz, CA, 455-466, 1987.

[165] Pfeffer, L.E. 1993. The Design and Control of a Two-Armed, Cooperating, Flexible-Drivetrain Robot System. Ph.D. thesis, Stanford University, Department of Mechanical Engineering.

[166] S. Quinlan. Efficient distance computation between non-convex objects. *Proc. IEEE Int. Conf. Robotics and Automation*, 3324-3330, San Diego, CA, 1994.

[167] Reif, J.H. and Sharir, M. 1985. Motion Planning in the Presence of Moving Obstacles. *Proc. FOCS.* New York:IEEE, pp. 144-154.

[168] E. Rimon and J.F. Canny. Construction of C-space roadmaps from local sensory data, What should the sensors look for? *Proc. IEEE Int. Conf. Robotics and Automation*, 117-123, San Diego, CA, 1994.

[169] E. Rimon and D.E. Koditschek. Exact robot navigation using artificial potential functions. *IEEE Tans. on Robotics and Automation*, 8:501-518, 1992.

[170] S.A. Schneider and R.H. Cannon, "Object Impedance Control for Cooperative Manipulation: Theory and Experimental Results," *IEEE Tr. Robotics and Automation*, 8(3), 1992, pp. 383-394.

[171] Schoppers, M.J., Representation and Automatic Synthesis of Reaction Plans, Ph.D. Dissertation, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1989.

[172] Shih, C., Lee, T., and Gruver, W.A. 1990. A Unified Approach for Robot Motion Planning With Moving Polyhedral Obstacles. *IEEE Tr. Systems, Man, and Cybernetics.* 20:903-315.

[173] P. Švestka. *A probabilistic approach to motion planning for car-like robots.* Tech. Rep. RUU-CS-93-18, Dept. Comput. Sci., Utrecht Univ., Utrecht, The Netherlands, April 1993.

[174] P. Švestka and M. Overmars. *Motion planning for car-like robots using a probabilistic learning approach.* To appear in *Int. Journal of Robotics Research*, 1995.

[175] P. Švestka and M. Overmars. Coordinated motion planning for multiple car-like robots using probabilistic roadmaps. *Proc. IEEE Int. Conf. Robotics and Automation*, Nagoya, Japan, 1631-1636, 1995.

[176] Takeda, H. and J.C. Latombe, Sensory Uncertainty Field for Mobile Robot Navigation, *Proc. of IEEE Int. Conf. on Robotics and Automation*, Nice, 2465-2472, May 1992.

[177] Taylor, R.H., *Synthesis of Manipulator Control Programs from Task-Level Specifications*, Ph.D. Dissertation, Department of Computer Science, Stanford University, 1976.

[178] Tournassoud, P., Lozano-Pérez, T., and Mazer, E. 1987. Regrasping. *Proc. IEEE Int. Conf. Robotics and Automation.* New York:IEEE, pp. 1924-1928.

[179] Wilfong, G. 1988. Motion Planning in the Presence of Movable Obstacles. *Proc. ACM Symp. on Computational Geometry.* New York:ACM, pp. 279-288.

[180] D. Williams and O. Khatib, "The Virtual Linkage: A Model of Internal Forces in Multi-Grasp Manipulation," *Proc. IEEE Int. Conf. Robotics and Automation*, Atlanta, GA, 1993.

[181] Zhang, Z. and Faugeras, O., A 3D World Model Builder with a Mobile Robot, *The Int. F. of Robotics Research*, 11(4):269-285, 1992.

[182] X. Zhu and K. Gupta. *On local minima and random search in robot motion planning.* Unpublished Tech. Report, Simon Fraser Univ., BC, Canada, 1993 (personal communication).